Argonne National Laboratory

A TRANSFORMATIONAL COMPONENT FOR PROGRAMMING LANGUAGE GRAMMAR

by

James M. Boyle

The facilities of Argonne National Laboratory are owned by the United States Government. Under the terms of a contract (W-31-109-Eng-38) between the U. S. Atomic Energy Commission, Argonne Universities Association and The University of Chicago, the University employs the staff and operates the Laboratory in accordance with policies and programs formulated, approved and reviewed by the Association.

MEMBERS OF ARGONNE UNIVERSITIES ASSOCIATION

The University of Arizona
Carnegie-Mellon University
Case Western Reserve University
The University of Chicago
University of Cincinnati
Illinois Institute of Technology
University of Illinois
Indiana University
Iowa State University
The University of Iowa
Kansas State University
The University of Kansas
Loyola University
Marquette University
Michigan State University
The University of Michigan
University of Minnesota
University of Missouri
Northwestern University
University of Notre Dame
The Ohio State University
Ohio University
The Pennsylvania State University
Purdue University
Saint Louis University
Southern Illinois University
University of Texas
Washington University
Wayne State University
The University of Wisconsin

Printed in the United States of America
Available from
Clearinghouse for Federal Scientific and Technical Information
National Bureau of Standards, U. S. Department of Commerce
Springfield, Virginia 22151
Price: Printed Copy $3.00; Microfiche $0.65

ANL-7690
Mathematics and Computers

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

A TRANSFORMATIONAL COMPONENT FOR PROGRAMMING LANGUAGE GRAMMAR

by

James M. Boyle

Applied Mathematics Division

Based on a thesis submitted in
partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the Graduate School of
Northwestern University

July 1970

TABLE OF CONTENTS

A TRANSFORMATIONAL COMPONENT FOR PROGRAMMING LANGUAGE GRAMMAR

By

James M. Boyle

ABSTRACT

The possibility of adapting the linguistic concept of transformational grammar for use with computer programming languages is investigated, and a formal definition and computer implementation of transformations are described. Two rather extensive examples of the usefulness of transformations as a definitional tool for programming languages are given.

A transformation consists of two patterns which describe tree structures in a context free phrase structure grammar. If the derivation tree of a string in the language of the grammar matches the first pattern, the string is to be transformed in a manner described by the second pattern. In the programming language adaptation of transformations, the transformed string is required to be a string in the same language as the original string, hence these transformations are called Intra-Grammatical Transformations (IGTs). In addition to terminal and nonterminal symbols of the grammar, patterns may contain "free symbols", which match any zero or more parts of a tree up to a match of the next non-free symbol in the pattern. The second pattern of an IGT may contain subtransformations, which are applied during construction of the transformed string and may employ items matched in the transformation containing them. Extensive built-in rules for controlling the sequence of application of groups of IGTs are also provided.

1. Introduction

During the past decade, one of the most significant problems in the theory of programming languages has been the search for suitable formalisms for programming language definition. Early success in the definition of syntax was achieved with the use of Backus-Naur Form grammar to define the syntax of Algol 60, but the development of a formalism for the definition of semantics has proceeded much more slowly, although its outlines are now becoming discernible.

A somewhat similar situation has existed in linguistics, where context free phrase structure grammars (later shown equivalent to BNF) were successfully used as the major component of grammars describing parts of the syntax of natural languages. To extend the descriptive power of these grammars, Chomsky introduced the transformational component. Productions in this new component, called transformations, are used to define certain semantically equivalent variations in the surface structure of sentences generated by the phrase structure component of the grammar.

The equivalence of BNF and context free grammar suggests the possibility of a programming-language analog for transformational grammar, which would be useful in defining those parts of a programming language which lie athwart the boundary between syntax and semantics: that is, those aspects which may be defined in terms of global changes in the form of a program which do not affect its meaning.

The goal of this investigation, then, is threefold: to show that transformational grammar can indeed be adapted for use with programming languages; that it can conveniently describe certain features of programming languages; and that a system for transformational grammar can be implemented on a computer so that such definitions can be tested.

1.1 Informal Introduction to Intra-Grammatical Transformations

This section is a step-by-step introduction to the transformational system for programming languages. It is intended to provide an intuitive understanding of the system which will be of assistance in following the rather complex formal definition given in section 2. The development proceeds by posing a sequence of increasingly sophisticated programming-language definitional problems and expounding transformational solutions to them; the discussion of these solutions introduces aspects of the notation and semantics for various parts of the transformational system. Throughout these discussions many italicized terms are defined informally (often by context); formal definitions of most of them are to be found in later sections.

All of the example transformations in this section define aspects of the semantics of *for* statements in Algol 60.* Many of them have direct counterparts among the *for* statement optimization transformations discussed in section 3.2; if a few of the remainder seem contrived, it is the result of a desire to present a unified set of examples.

Because the transformational system for programming languages is strongly rooted in the concept of transformational grammar as used in linguistics, I will first discuss the linguistic concept of transformational grammar briefly, noting some similarities and differences

---

*Familiarity with the Revised Report on Algol 60 [30] is assumed. A slightly modified Algol syntax (used for the examples of section 3) is included as section 6.1.

between it and the transformations introduced here.

As mentioned earlier, the concept of transformational grammar was introduced by Chomsky in terms of a transformational component to supplement the phrase structure component of natural language grammars. The transformational component makes two important contributions to the elegance and simplicity of a natural language grammar: it largely overcomes the inadequacy of phrase structure grammars alone for generating natural languages, while still permitting retention of the relatively simple phrase structure component of the grammar; and it unifies the description of groups of related (usually semantically equivalent) constructions (e.g., the active and passive forms of a sentence) by permitting the phrase structure component to generate but a single *parse tree*, or deep structure, for the group, with differences in surface structure among members of the group being accounted for by the transformational component.

As used in natural language grammars, a transformational production consists of two principal parts: the *structural description (SD)* and the *structural change (SC)*. To these are sometimes added certain additional restrictions, such as equality restrictions. The structural description is a sequence of terminal and nonterminal symbols of the phrase structure grammar; if the parse tree of a particular sentence in the grammar is described by the SD (essentially, if a line can be drawn in the parse tree through the sequence of nodes given by the SD without cutting any branches of the parse tree) the transformation is said to *apply*. If a transformational applies, the

structural change is interpreted as describing (by an obvious use of subscripted variables) a rearrangement of the terminal strings of the subtrees matched by symbols in the SD. Structural descriptions and structural changes may also contain variable (or *indefinite*) symbols, which match any zero or more nodes up to the first occurrence of the following definite symbol.

Perhaps the most classic linguistic transformation is the passive transformation for English described by Chomsky (quoted in Bach [2], p. 62). It is*.

Passive, optional:

Structural description: $NP - Aux - V_{tr} - NP$

Structural change: $X_1 - X_2 - X_3 - X_4$

$\rightarrow X_4 - X_2 + be + en - X_3 - by + X_1$

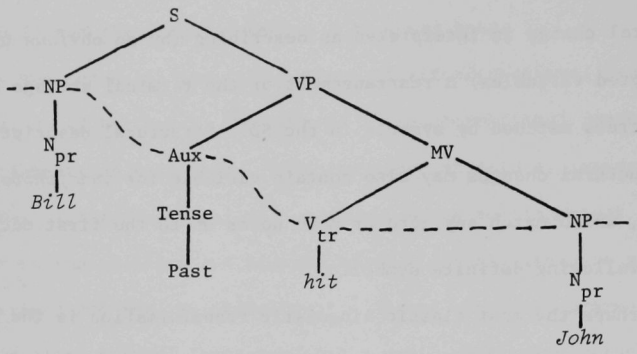
or, in a slightly more economical notation:

$NP - Aux - V_{tr} - NP' \rightarrow NP' - Aux + be + en - V_{tr} - by + NP$

This transformation may (optionally) be applied to the parse tree (assuming a suitable phrase structure grammar) of an English sentence in the active voice to produce its passive counterpart. For example, consider the parse tree:

---

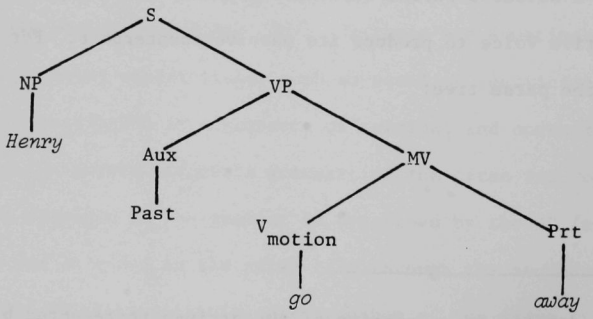
*Here "-" denotes concatenation of the strings represented by nonterminal symbols; "+" denotes concatenation of terminal symbols or of a terminal symbol and the string represented by a nonterminal symbol.



that is, *Bill hit John*. The dashed line indicates that the passive transformation does indeed apply, and the structural change specifies that the corresponding passive sentence is obtained by interchanging the strings represented by the first and fourth symbols of the match and inserting some terminal symbols, giving:

John + Past + *be* + *en* + *hit* + *by* + *Bill*

i.e., *John was hit by Bill*. On the other hand, for the tree:



no match is possible with the SD of this transformation, and it does not apply.

The above discussion has carefully avoided the question of assigning a parse tree to the transformed sentence, which obviously would be required if it were necessary to apply another transformation to it. This is still somewhat an open question and has not been treated adequately in the open literature. (The principal discussion is in unpublished notes of Chomsky. Bach [2] provides a brief intuitive discussion, and Petrick [31] discusses the problem from the analytic, rather than generative, point of view.)

In adapting the concept of transformational grammar to programming languages, I have overcome the difficulty of specifying the parse tree for the transformed string by viewing the written, or *concrete*, forms of the SD and SC of a transformation as convenient representations for their *abstract* forms, which are tree structures (*sd* and *sc trees*) similar to parse trees. To apply such a transformation its sd tree is matched against the (upper part of the) parse tree to be transformed, and if it matches, the sc tree is used to construct the (upper part of the) parse tree of the transformed string. Suitable restrictions on the sc tree insure that the transformed string has a parse tree in the same grammar as the original string, hence the name *intragrammatical transformations (IGTs)*. Transformations constructed in this manner are also much more nearly invertible (e.g., for use in generation rather than analysis) than those customarily employed in linguistics.

(1.1)

To introduce the notation for the concrete forms of IGTs, I will cast the above passive transformation as an IGT; its interpretation remains unchanged:

S{'SD'

NP"1" Aux"1" V_{tr}"1" NP"2"

==>

NP"2" Aux"1" *be en* V_{tr}"1" *by* NP"1"

'SC'}

1.1.1 Elementary Intra-Grammatical Transformations

To begin this discussion of elementary aspects of IGTs, consider the definition of the simplest form of for statement in Algol 60, i.e., a for statement whose <for list> is simply an <arithmetic expression> (cf. sections 4.6.1 and 4.6.4.1 of the Revised Report [30]). The concrete form of an IGT which converts this form of for statement into its equivalent Algol statements is:

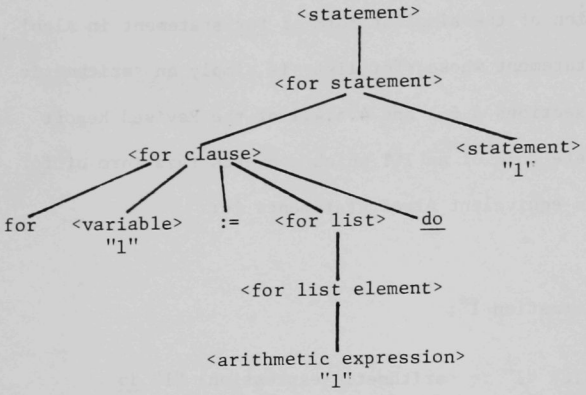
```
<statement>{
   comment Transformation 1*;
   'SD'
      for <variable> "1" := <arithmetic expression> "1" do
         <statement> "1"
   ==>
      begin <variable> "1" := <arithmetic expression> "1";
         <statement> "1"
      end
   'SC'}
```

The structural description of this IGT is the string of symbols found between the delimiters 'SD' and ==> while the structural change is delimited by ==> and 'SC'. The initial symbol "<statement>" is the *dominating symbol* of the IGT and indicates in an obvious manner the head symbol of the *sd tree* and *sc tree* which constitute the abstract

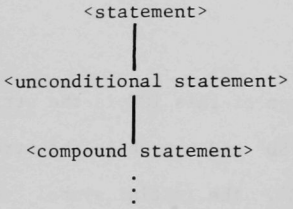
---

*The Algol 60 convention of delimiting comments by "comment" and ";" is followed; comments may be inserted anywhere in a transformation.

form of the transformation. In this case, the sd tree (according to the Algol 60 grammar) is:



The sc tree is constructed according to the grammar in a similar manner, and begins:



Throughout the remainder of this discussion transformations will always be written in their concrete form, but it is important to keep in mind that they are actually tree structures.

In the SC, the *variable* consisting of a symbol and its *index* (e.g. <statement> "1") indicates where in the transformed tree the

subtree matched by an occurrence of that variable in the SD is to be placed. For example, Transformation 1 applies to the statement:

```
for i := 2 do a[i] := 2×i                                    (1)
```

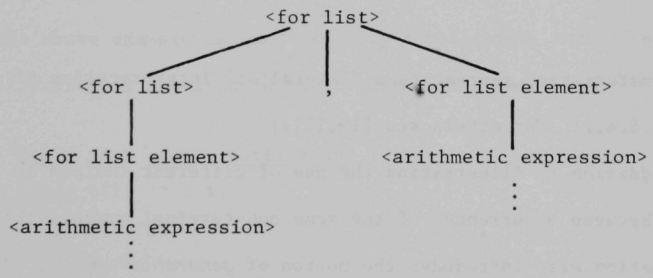
to produce the statement:

```
begin i := 2;
   a[i] := 2×i                                          (T1(1))
end
```

Note, however, that it does not apply to the statement:

```
for i := 1, 2 do a[i] := 2×i                                 (2)
```

since the <for list> subtree of this statement is:



which is not of the same form as the <for list> subtree of the sd tree.

A somewhat more interesting IGT is the one defining the meaning of the for statement with a step-until element. Modifying the definition of section 4.6.4.2 of the Revised Report slightly, this may be written:

```
<statement>{
   comment Transformation 2;
   'SD'
      for <variable> "1" := <arithmetic expression> "1" step
          <arithmetic expression> "2" until <arithemtic expression> "3"
              do <statement> "1"
   ==>
      begin <variable> "1" := <arithmetic expression> "1";
      <label> "1":
         if (<variable> "1" - (<arithmetic expression> "3"))
             × sign (<arithmetic expression> "2") ≤ 0 then
         begin <statement> "1";
             <variable> "1" :=  <variable> "1"
                + (<arithmetic expression> "2");
             go to <label> "1"
         end
      end
   'SC'}
```

(This transformation represents a "literalist" interpretation of section 4.6.4.2. For others see [19,21].)

In addition to illustrating the use of different indices to distinguish between occurrences of the same non-terminal symbol, this transformation also introduces the notion of *generable symbol*. A generable symbol is a variable, such as <label> "1" in Transformation 2, which appears in the SC of a transformation but not in the SD. The use of such a variable in the SC occasions the generation of a subtree headed by that symbol, which is then associated with that variable just as if it had occurred in the SD. Of course, in this example, care must be taken to generate a <label> which does not occur elsewhere in

the program being transformed.

Transformation 2 applied to the string:

```
for i := 1 step 2 until n do a[i] := 2×i                    (3)
```

transforms it into:

```
begin i := 1;
label7:
   if (i - (n)) × sign (2) ≤ 0 then                    (T2(3))
   begin a[i] := 2×i;
      i := i + (2);
      go to label7
   end
end
```

Now one may very well complain that the transformed statement in the above example is unnecessarily complicated, even though this complication is required in the transformation of:

```
for i := 0 step j until k do
begin a[i] := 2×i;                                          (4)
   j := -2×j;
   k := -k
end
```

(which is presumably to be executed for i = 0, -2, +2, -6 when initially j = 1, k = 6). More appropriate for (3) would be:

```
<statement>{
   comment Transformation 3;
   'SD'
      for <variable> "1" := <arithmetic expression> "1" step
         <unsigned integer> "1" until <arithmetic expression> "2" do
         <statement> "1"
   ==>
      begin <variable> "1" := <arithmetic expression> "1";
      <label> "1":
         if <variable> "1" ≤ <arithmetic expression> "2" then
         begin <statement> "1";
            <variable> "1" := <variable> "1" + <unsigned integer> "1";
            go to <label> "1"
         end
      end
   'SC'}
```

The result of Transformation 3 applied to (3) is:

```
begin i := 1;
label8:
   if i ≤ n then
   begin a[i] := 2×i;                                    (T3(3))
      i := i + 2;
      go to label8
   end
end
```

while it does not apply to (4), since j is not an <unsigned integer>.

The introduction of Transformation 3 raises two questions: When two or more transformations are present, in what order should they be applied? What further improvements can be made in the transformed

<statement> for simple for statements like (3)? The first of these questions is discussed in the following section; the other is discussed in section 1.1.3.

1.1.2 Sequencing Rules for Intra-Grammatical Transformations

For Transformation 2 and 3 of section 1.1.1 the question of the order in which they should be applied can be answered quite easily: apply Transformation 3 and then Transformation 2, since Transformation 3 applies to a more restricted class of statements than does Transformation 2. Thus for statement (3), Transformation 3 would apply, and clearly Transformation 2 will not apply to $T_3(3)$, whereas for statement (2), Transformation 3 will not apply, permitting the application of Transformation 2. A further example will illustrate that this simple sequencing rule must be extended in order to be useful, however.

Consider constructing transformations to define the semantics of multiple <for list element>s. A straightforward solution to this problem is to replace a for statement having n <for list element>s by a compound statement consisting of n for statements, the i^{th} of which contains only the i^{th} <for list element> of the original (cf. [6]). Such an approach involves n replications of the controlled statement, however, and in practice it is usually more efficient to construct a procedure from the controlled statement and call it in each of the n for statements (cf. Grau, Hill, and Langmaack [15], p. 96). The following transformations express this definition:

```
<statement>{
   comment Transformation 4.1;
   'SD'
      for <variable> "1" := <for list> "1", <for list element> "1" do
         <procedure statement> "1"
   ==>
      begin for <variable> "1" := <for list> "1" do
            <procedure statement> "1";
         for <variable> "1" := <for list element> "1" do
            <procedure statement> "1"
      end
   'SC'
   comment Transformation 4.2;
   'SD'
      for <variable> "1" := <for list> "1", <for list element> "1" do
         <statement> "1"
   ==>
      begin procedure <identifier> "1"; <statement> "1";
         for <variable> "1" := <for list> "1" do <identifier> "1";
         for <variable> "1" := <for list element> "1" do
            <identifier>  "1"
      end
   'SC'}
```

Note that when one or more transformations have the same dominating symbol they are all written within the braces indicating the scope of the dominating symbol. If the dominating symbol is, e.g., <statement>, such a construction is called a *<statement>-transformation list.*

Applied to (2) of section 1.1.1, the above transformations produce the transformed statement:

```
begin procedure identifier3; a[i] := 2×i;
   for i := 1 do identifier3;                    (T4.2(2))
   for i := 2 do identifier3
end
```

However, consider:

```
for i := 1, 2, 3 do P(i)                         (5)
```

Transformation 4.1 applies to give:

```
begin for i := 1, 2 do P(i);
   for i := 3 do P(i)                            (T4.1(5))
end
```

Clearly what is necessary to complete the transformation of $(T_{4.1}(5))$ is to require the (recursive) *reapplication* of Transformations List 4, i.e., it must be reapplied to any <statement>s introduced as a result of transformation. (In practice, it is efficient to do this during the construction of the parse tree specified by the SC.) Then (5), when fully transformed, becomes:

```
begin
   begin for i := 1 do P(i);
      for i := 2 do P(i)                         (T4.1(T4.1(5))
   end;
   for i := 3 do P(i)
end
```

The sequencing rule for a transformation list is thus: Attempt to apply the transformations in order. If none applies, the result is the original tree. If some transformation does apply, attempt

to apply the transformation list to all subtrees constructed during the construction specified by the SC (i.e., apply the transformation list *recursively* to the transformed parse tree). One should note that this sequencing rule is essentially the appropriate generalization for tree structures of the sequencing rule for Markov algorithms without terminal productions (cf. [29], chapter 5). However, it is perhaps most properly viewed as a consequence of the general rule for the application of transformation lists to the parse tree of a <program>, discussed below.

It is reasonable that the application of, say, a <statement>-transformation list to the parse tree of a <program> should mean that the transformation list is applied to each subtree of the parse tree headed by <statement>. However, for symbols such as <statement> in Algol 60, where a <statement> subtree may itself contain <statement> subtrees, it is necessary to specify whether transformation lists are to be applied in a top-down or bottom-up traverse of the parse tree.

There does not seem to be any strong theoretical reason to choose one of these orders over the other, but it is an empirical fact (cf. the examples of section 3) that bottom-up order is an advantageous choice. This choice is perhaps partly justified because it enables one to write, for example, a <for statement> transformation list with the knowledge that when it applies to a <for statement> which originally contained other <for statement>s, the contained <for statement>s have already been fully transformed according to

that transformation list. Of course, for this statement to be valid, the <for statement>-transformation list must be reapplied (in bottom-up order) to any <for statement>s generated in the construction of the transformed subtree. This reapplication rule, together with the fact that every transformed subtree is headed by the same symbol as was the subtree it replaces, naturally results in what appears to be generalized Markov sequencing of transformation lists.

The choice of bottom-up sequencing is also partly justified by the fact that a kind of top-down sequencing rule applies to the matching of *indefinites* (cf. section 1.1.3); thus both kinds of sequencing are available. Finally, the choice of bottom-up sequencing coincides with that recommended by Chomsky for natural language transformational grammars in his more recent thinking (cf. [10], pp. 134, 135, 143).

The solution of the following problem not only illustrates one advantage of bottom-up sequencing, but also introduces the concept of transformation set. Consider again Transformation 3 of section 1.1.1. As written, it will not apply to:

```
for i := 1 step +2 until n do a[i] := 2×i                    (6)
```

(because +2 is not an <unsigned integer>) even though this statement could just as well be transformed in a manner analogous to $T_3(3)$. One possibility, of course, is to write another transformation identical to Transformation 3 in every respect except with + <unsigned integer> "1" replacing <unsigned integer> "1" in the SD.

A simpler and more general solution, however, is to combine the following <simple arithmetic expression>-transformation list:

```
<simple arithmetic expression>{
   comment Transformation 5;
   'SD'
      + <term> "1"
   ==>
      <term> "1"
   'SC'}
```

with the <statement>-transformation list containing Transformation 3. Such a group of one or more transformation lists, no two having the same dominating symbol, is called a *transformation set*. The sequencing rule is then modified to read as follows: Given a transformation set and the parse tree of a <program>, attempt to apply the transformation set (recursively) to each subtree of the parse tree, traversed in bottom-up, left-to-right order. By application of a transformation set to a subtree of a parse tree is meant: If the transformation set contains a transformation list whose dominating symbol is the same as the head symbol of the subtree, apply that transformation list to the subtree; otherwise do nothing. (Note that the order of the transformation lists comprising a transformation set is immaterial; this is reflected by the name "transformation set". The name "transformation list", on the other hand, reflects the significance of the order of the transformations comprising it.)

The bottom-up sequencing rule for transformation sets thus

guarantees that by the time Transformation 3 is applied to (6), the step element +2 will have been replaced by simply 2.

There remains one further combination of transformations to be defined: the *transformation set sequence*. As its name implies, it is a sequence of transformation sets. The rule for applying a transformation set sequence to a <program> parse tree is quite simple: Apply the first transformation set of the sequence to the parse tree (according to the rule given above), and then apply each succeeding transformation set to the transformed <program> parse tree produced by its predecessor; the result of applying the sequence is the transformed <program> parse tree produced by the last transformation set. An example of the use of transformation set sequences is the sequence consisting of the Identifier Denotation Transformation Set and the For Statement Optimization Transformation Set described in section 3.

To help the reader verify his understanding of the concepts introduced so far, I conclude this section with the description of a transformation set sequence (consisting of a single <statement>-transformation list) which, applied to a <program> parse tree, completely eliminates all for statements in it, substituting for them their definitions in terms of simpler Algol statements (without, however, making any attempt at optimization):

```
{<statement>{
   comment Transformation 6.1. (This is transformation 2);
   'SD'
      for <variable> "1" := <arithmetic expression> "1" step
         <arithmetic expression> "2" until <arithmetic expression> "3"
            do
         <statement> "1"
   ==>
      begin <variable> "1" := <arithmetic expression> "1";
      <label> "1":
         if (<variable> "1" - (<arithmetic expression> "3"))
            × sign (<arithmetic expression> "2") ≤ 0 then
         begin <statement> "1";
            <variable> "1" := <variable> "1"
               + (<arithmetic expression> "2");
            go to <label> "1"
         end
      end
   'SC'
   comment Transformation 6.2;
   'SD'
      for <variable> "1" := <arithmetic expression> "1"
            while <Boolean expression> "1" do
         <statement> "1"
   ==>
      begin
      <label> "1": <variable> "1" := <arithmetic expression> "1";
         if <Boolean expression> "1" then
         begin <statement> "1";
            go to <label> "1"
         end
      end
   'SC'
```

```
comment Transformation 6.3. (This is Transformation 1);
'SD'
   for <variable> "1" := <arithmetic expression> "1" do
      <statement> "1"
==>
   begin <variable> "1" := <arithmetic expression> "1";
      <statement> "1"
   end
'SC'
comment Transformation 6.4. (This is Transformation 4.1);
'SD'
   for <variable> "1" := <for list> "1", <for list element> "1" do
      <procedure statement> "1"
==>
   begin for <variable> "1" := <for list> "1" do
         <procedure statement> "1";
      for <variable> "1" := <for list element> "1" do
         <procedure statement> "1"
   end
'SC'
comment Transformation 6.5. (This is Transformation 4.2);
'SD'
   for <variable> "1" := <for list> "1", <for list element> "1" do
      <statement> "1"
==>
   begin procedure <identifier> "1"; <statement> "1";
      for <variable> "1" := <for list> "1" do <identifier> "1";
      for <variable> "1" := <for list element> "1" do
         <identifier> "1"
   end
'SC'}}
```

One may verify that application of this transformation set sequence to a <program> parse tree containing the <statement> subtree:

```
for i := 1, i+1 while a[i] ≠ 0 do
for j := 1 step 1 until n do b[i,j] := a[i]×j                    (7)
```

yields $(T_{6.2}(T_{6.3}(T_{6.5}(T_{6.1}(7)))))$:

```
begin procedure identifier2;
   begin j := 1;
   label1:
      if (j-(n)) × sign (1) ≤ 0 then
      begin b[i,j] := a[i]×j;
         j := j+(1); go to label 1
      end
   end;
   begin i := 1; identifier2 end;
   begin label3: i := i+1;
      if a[i] ≠ 0 then
      begin identifier2; go to label3 end
   end
end
```

1.1.3 Complex Intra-Grammatical Transformations: Indefinites

Consider again the second question raised at the end of section 1.1.1: What further improvements can be made in the transformed statement for simple for statements? Most of the improvements one would like to make consist of moving some computation which is invariant with respect to the loop to a position outside the loop, e.g., moving the calculation of the until-expression outside the loop. The chief problem, then, is to determine invariance; this consists primarily in checking that the computation does not contain the loop variable, nor a variable assigned in the loop, and that the loop contains no procedure calls, etc., which could result in "hidden" changes in variables in the computation being checked (cf. section 3.2 for a more detailed discussion of this problem).

Such checking operations require some convenient means of searching a subtree for any occurrence of some other (usually previously matched) subtree. To enable IGTs to express these searches, indefinite nodes, or *indefinites*, are introduced. Consider the problem of searching the until-expression of a for statement for an occurrence of the loop variable; using an indefinite this may be expressed:

```
<statement>{
  comment Transformation 7;
  'SD'
    for <variable> "1" := <arithmetic expression> "1"
      step <arithmetic expression> "2" until
      <arithmetic expression> "3" { ? <variable> "1" ? }
```

```
      do <statement> "1"
==>
   comment Same SC as Transformation 2;
'SC'}
```

Here <arithmetic expression> "3" is an indefinite, and has an *indefinite list* consisting of { ? <variable> "1" ? }.* The symbol "?" is called the *free symbol*. When used in the SD of a transformation, a free symbol matches that part of a subtree (possibly nothing at all) up to an appropriate occurrence of the next non-free symbol. When used in the SC of a transformation a free symbol simply acts as a placeholder for the part of the tree in matched in the SD.

Three obvious restrictions are placed an indefinites: (1) no two free symbols may be adjacent, since there would then be no way to determine where one ended and the next began: (2) if a given variable is indefinite in the SC, it must be indefinite exactly once in the SD, to insure that the part of the tree matched by the free symbols can be found; and (3) the head symbols of the subtrees and the free symbols in the indefinite lists must be the same, and occur in the same order, in the SD and SC, to insure that a correct tree is obtained when copying the indefinite.

---

*Non-indefinite lists, which contain no occurrences of "?", are also permitted, e.g.,

<subscript expression> "1" {<unsigned integer> "1" × <variable> "1"
<adding operator> "1" <unsigned integer> "2"}

They are useful for giving names to subtrees which must have a certain substructure, and for obtaining compliance with restriction (3) of the following paragraph.

Some further examples will help to illustrate the meaning of indefinites. It should be clear that Transformation 7 applies to:

```
for i := 1 step 1 until 10-i do
    a[i] := 2×i                                    (8)
```

and also to:

```
for i := 1 step 1 until n×i-7 do
    a[i] := 2×i                                    (9)
```

but does not apply to, e.g., (3) or (4) of section 1.1.1. (Note that when the same variable occurs more than once in the SD of a transformation all occurrences must match equal subtrees.)

Had the indefinite of Transformation 7 been written:

<arithmetic expression> "3" { ? <variable> "1"}

it would apply to (8) but not (9), i.e., if there is no free symbol to the right (left) of a non-free symbol, the subtree matched by the non-free symbol must not have any subtrees to the right (left) of it. Similarly, two adjacent non-free symbols must match "adjacent" subtrees (i.e., there must be no subtrees between them). For example, the transformation:

```
<statement>{
   comment Transformation 8;
   'SD'
      for <variable> "1" := <arithmetic expression> "1" step
         <arithmetic expression> "2" until <arithmetic expression> "3"
             do
         <statement> "1" { ? <variable> "1" := ? }
   ==>
      comment Same SC as Transformation 2;
   'SC'}
```

applies to:

```
for i := 1 step 1 until n do
begin a[i] := 2×i;                                          (10)
   i := i+1
end
```

but not to:

```
for i := 1 step 1 until n do                                (11)
begin a[i] := 2×i end
```

A more complex use of indefinites is illustrated by a transformation which determines whether the until-expression of a for statement contains a variable assigned in the controlled statement:

```
<statement>{
   comment Transformation 9;
   'SD'
      for <variable> "1" := <arithmetic expression> "1" step
         <arithmetic expression> "2" until
         <arithmetic expression> "3" { ? <identifier> "1" ? }
         do <statement> "1"
         { ? <variable> { <identifier> "1" ? } := ? }
   ==>
      comment Same SC as Transformation 3;
   'SC'}
```

In this transformation the success or failure of the match of <identifier> "1" in the second indefinite influences the match of <identifier> "1" in the first indefinite.

Transformations 7, 8, and 9 are but a few of several which are required to determine whether a for statement can contain invariant calculations. Each of these transformations has the property that if it applies to a for statement, certain calculations are not invariant and the for statement is to be transformed in the most general way; thus all of the transformations have essentially the same SC.

When this is the case, the transformations are simplified by having each of them *mark* the for statement in a way which prevents the transformation from reapplying; a single transformation placed late in the list then converts all marked for statements to the general form described by Transformation 2. Perhaps the most convenient *markers* are obtained by introducing "synonyms" for terminal symbols; in this case for1, for2, etc., are appropriate. This method is

especially flexible if a name can be given to the whole collection of synonyms for a given terminal symbol, either by replacing, say, the symbol for in the Algol grammar by <for symbol> and adding a production <for symbol> ::= for | for1 | for2 or else by letting an indexed terminal symbol in a transformation match either itself or any of its synonyms; this latter approach is used in the transformations discussed in section 3. Using markers, the SC of Transformations 7, 8, and 9 would be (T'_7, T'_8, and T'_9):

```
for1 <variable> "1" := <arithmetic expression> "1" step
   <arithmetic expression> "2" until <arithmetic expression> "3"
      do
   <statement> "1"
```

while the beginning of the SD of Transformation 2' (which must occur after Transformations 7', 8', and 9' in the list) would be:

```
<for symbol> <variable> "1" := ...
```

(or for "1" <variable> "1" := ..., if the second approach above is employed).

Another use of a marker is illustrated by a transformation which looks for a procedure or function procedure call anywhere in a for statement after step:

```
<statement>{
   comment Transformation 10;
   'SD'
      <for statement> "1"
      {<for symbol> {for} <variable> "1" := <arithmetic expression> "1"
         step ? <procedure identifier> "1" ?}
   ==>
      <for statement> "1"
      {<for symbol> {forl} <variable> "1" := <arithmetic expression> "1"
         step ? <procedure identifier> "1" ?}
   'SC'}
```

Note that Transformation 10 obeys restriction 3 mentioned earlier even though for and forl differ, since in both cases the head symbol of the subtree containing them is <for symbol>.

Transformation 10 applies to:

```
for i := 1 step f(x) until n do
   a[i] := 2×i                                                    (12)
```

and

```
for i := 1 step 1 until n do
begin a[i] := 2×i; p(x,y,i) end                                   (13)
```

among others.

If no transformation like 7', 8', 9' or 10 applies to a given for statement, so that the until-expression is known to be invariant with respect to the loop, it may be transformed according to Transformation 11 (which must, of course, precede Transformation 2'):

```
<statement>{
   comment Transformation 11;
   'SD'
      for <variable> "1" := <arithmetic expression> "1" step
         <unsigned integer> "1" until <arithmetic expression> "2" do
         <statement> "1"
   ==>
      begin integer <identifier> "1";
          <variable> "1" := <arithmetic expression> "1";
          <identifier> "1" := <arithmetic expression> "2";
      <label> "1": if <variable> "1" ≤ <identifier> "1" then
                   begin <statement> "1";
                         <variable> "1" := <variable> "1"
                               + <unsigned integer> "1";
                         go to <label> "1"
                   end
      end
   'SC'}
```

Any for statements with step-until elements to which Transformation 11 does not apply would then be transformed by Transformation 2'.

The chief remaining class of computations some of which may be invariant (or linear) with respect to the loop is the calculation of subscripts; searching subscript expressions for occurrences of variables assigned in the loop will be considered in the following section.

1.1.4 Complex Intra-Grammatical Transformations: Subtransformations

The last feature of IGTs remaining to be described is *subtransformations* (technically, *subtransformation sequences*). As the name implies, a subtransformation is basically a transformation which occurs within the SC of another transformation. The use of subtransformations considerably increases the elegance of many transformational definitions, as is perhaps best illustrated by the identifier denotation transformations described in section 3.1.

The simplicity and convenience of subtransformations derives primarily from two aspects of their semantics: their ability to employ indexed symbols matched in the SD (or generated in the SC) of the transformation(s) in which they occur; and their more limited, non-recursive sequencing rules. These concepts can be illustrated by considering a transformation which examines the controlled statement of a for statement and marks all subscripted variables which contain a variable assigned in the controlled statement by replacing "[" with the marker "$[_1$":

```
<for statement>{
   comment Transformation 12;
   'SD'
      for <variable> "1" := <for clause> "1" { ? step ? } do
         <statement> "1"
         { ?  variable  "2" {<identifier> "1" ? }
         <assign symbol> {:=} ? }
   ==>
      for <variable> "1" := <for clause> "1" do
         <statement> "1" { ? <variable> "2" <assign symbol> {..=} ? }
      {comment Transformation 11.A.1;
      'SD'
         <statement> "11"
         { ? <subscripted variable>
               { <identifier> "2" <left bracket> {[} ?
                 <identifier> "1" ? ] } ?
         }
      ==>
         <statement> "11"
         { ? <subscripted variable>
               { <identifier> "2" <left bracket> {[1} ?
                 <identifier> "1" ? ] } ?
         }
      'SC'
      }
   'SC'}
```

In this transformation, the main transformation applies to any for statement with a step-until element and at least one variable followed by ":=" in its controlled statement, and in constructing the transformed controlled statement, it marks that variable as having been investigated by replacing ":=" by "..=". (Thus the

main transformation eventually will cease to apply.) Furthermore, after the transformed controlled statement has been constructed, the subtransformation is applied to it. Since the subtransformation has available to it the identifier of the variable assigned (<identifier> "1"), it can mark any subscripted variables containing that identifier in their <subscript list>. Insertion of the marker "$[_1$" not only guarantees termination of the subtransformation but also identifies the subscripted variable as non-linear and non-constant to a later optimizing transformation (cf. section 3.2). Note that the construction:

<variable> "2" {<identifier> "1" ? } :=

is used to determine <identifier> so that the identifier of a subscripted variable, rather than the entire subscripted variable, is obtained.

Consider the statement:

```
for i := 1 step 1 until n do
begin j := n-i;                                    (14)
   c[a[k]] := i;
   a[i] := b[j]
end
```

One complete application of Transformation 12 produces ($T_{12}(14)$):

```
for i := 1 step 1 until n do
begin j ..= n-i;
   c[a[k]] := i;
   a[i] := b[1j]
end
```

(Here <identifier> "1" was "j".) The transformation reapplies, giving ($T_{12}(T_{12}(14))$):

```
for i := 1 step 1 until n do
begin j ..= n-i;
   c[a[k]] ..= i;
   a[i] := b[1j]
end
```

(In this case, <identifier> "1" was "c", and the subtransformation did not apply.) Finally, it applies a third time to give ($T_{12}(T_{12}(T_{12}(14)))$):

```
for i := 1 step 1 until n do
begin j ..= n-i;
   c[1a[k]] ..= i;
   a[i] ..= b[1j]
end
```

Thus c[a[k]] and b[j] have been marked as neither invariant nor linear.

Note that the *dominating symbol* for Subtransformation 12.A.1 is <statement> "1"; the general rule for determining the dominating symbol of a subtransformation is that it is the symbol immediately preceding the "{" symbol introducing the subtransformation list,

unless that symbol is "}", in which event the dominating symbol is the dominating symbol of the "{" symbol matching the "}" symbol. In particular, one may have more than one subtransformation list (i.e., a subtransformation sequence of length greater than one) with the same dominating symbol. In this case the subtransformation lists making up the sequence are applied one after another in exact analogy to the rule for applying the transformation sets of a transformation set sequence. The scope rules for indexed symbols in this case are analogous to the scope rules for identifiers in Algol 60: a subtransformation "knows" all the indexed symbols in transformations containing it, but, of course, it does not have access to those in other subtransformations in its list or sequence (cf. sections 2.2.1 and 2.3.2).

The sequencing rules for subtransformation lists differ from those for transformation lists in that a subtransformation list is applied only to that subtree of the parse tree which is constructed from its dominating symbol (i.e., no traverse is made of this subtree). It continues to be applied to this subtree until no transformation in the list applies. The reasons for adopting this sequencing rule are most empirical; they are discussed further in sections 2.3.2 and 4.1. No further examples of subtransformations will be given here, as the detailed examples of section 3 contain numerous applications of them.

This discussion brings to a close the informal introduction to intra-grammatical transformations. One should now be prepared

to study their formal definition in section 2, and to understand the examples of section 3. Before the formal definition of IGTs is introduced, however, their relation to earlier work will be briefly discussed in the following section.

1.2 Relation of Intra-Grammatical Transformations to Previous Work

As discussed above, IGTs are based on natural language transformational grammars, especially as defined by Chomsky [10, 11]. They may thus be considered both in relation to work in linguistics on transformational grammar, and in relation to previous similar work in programming language theory.

Much of the research on transformational grammar in linguistics has been concerned with its use as a tool for describing various aspects of natural languages. This work is relevant to IGTs only to the extent that it has provided feedback concerning the adequacy of transformational grammar to its linguistic tasks; a number of modifications suggested by this work are discussed in [10].

There are, however, a few computer implementations of transformational grammars as used in linguistics. As mentioned above, Petrick [31] has implemented a transformational analysis system with the goal of inverting a transformational grammar in order to use it to analyze the sentences it generates. Such a system immediately encounters two problems: First, no general algorithm for inverting transformations is known (cf. Hays [18], cha. 8). Second, it is very difficult to assign sufficient tree structure to an input string to determine when an inverted transformation applies, since the only phrase structure grammar available to such a system is the one which generates the "deep structure" of the strings, i.e., their tree structure before any (generative) transformations have applied to

them. Petrick considers and rejects the introduction of a "covering grammar" (essentially the intra-grammatical restriction imposed on IGTs), since such a grammar is not usually available or derivable for natural language transformational grammars, and attacks the problem by alternating application of phrase-structure parsing and transformational analysis. Unfortunately this technique is slow and only applies to a subset of all transformational grammars.

Friedman [14] has also described a computer system for transformational grammar. This system was designed with the goal of providing the linguist with a comprehensive generative transformational system which would assist him in formulating and "debugging" transformational grammars for natural languages. Like IGTs, it is a generalization and formalization of the concept of transformational grammar of Chomsky [10]. Except for the introduction of some quite powerful control mechanisms, most of the generalization has been introduced to better adapt the system to its linguistic tasks and at present has little application to programming languages. The basically generative orientation of the system also limits its usefulness with programming languages, except perhaps for its intended use as a tool to assist in the design of a (generative) grammar for a specific language.

Several authors in the field of programming languages have described systems similar to grammatical transformations, although apparently no one has emphasized the analogy. These systems are frequently described by the term "generalized Markov algorithms,"

since they typically employ the matching and sequencing rules for Markov algorithms (cf. [29] for an introductory discussion of Markov algorithms). Perhaps the most important among these systems are those of Caracciolo (cf., for example [9]; the details of Caracciolo's work have not been published in the open literature), de Bakker [4], and Iturriaga [20]. These systems are described in section 3.2 of a recent excellent survey article by de Bakker [3], and will not be further described here. Iturriaga claims that his system differs from the others in that the transformations operate on tree structures (as do IGTs) rather than on strings. Three other systems incorporating tree transformations but not discussed by de Bakker are those of McIntosh (not published; discussed informally by Brody [7]), Reynolds [34], and Maruyama [24].

The system of IGTs introduced here differs from the above systems in three principal ways: first, in the use of FREE symbols and the method of matching indefinites in the SD and transmitting their tree structure to the SC; second, in the introduction of subtransformations and their scope rules; and third, in its emphasis on built-in sequencing mechanisms rather than *ad hoc* methods, such as each transformation explicitly nominating its successor. Furthermore, the examples of IGTs discussed in section 3, especially the for statement optimization transformation set, represent new applications of transformational methods to the description of programming languages.

2. Intra-Grammatical Transformations

Section 2 comprises the formal definition of intra-grammatical transformations over a context free phrase structure grammar G, the proof of a few theorems characterizing some formal properties of these transformations, and the discussion of a computer implementation of an intra-grammatical transformation system.

The formal definition of IGTs consists of a specification of their syntax and semantics. The syntax specification describes both the abstract form of IGTs--the tree structures which represent the SDs and SCs of transformations--and the concrete form of IGTs--a linear written representation of the abstract form. The semantic specification describes the conditions under which a parse tree p_2 is the result of applying a transformation to another parse tree p_1. One should note that this semantic specification does not describe how to compute p_2, but only how to determine whether p_2 is the transform of p_1; the LISP functions described in section 2.5 define the computation of p_2 from p_1 and the transformation.

The notation and some basic definitions used in defining the syntax and semantics of IGTs will be introduced in the following subsection. It also defines the parse trees which represent the programs being transformed.

2.1 Notation and Definitions

The notation used to define the syntax and semantics of IGTs is that of the "Method and Notation for the Formal Definition of Programming Languages" [23], hereafter referred to as the "Vienna Report." Familiarity with sections 1.3, 2, 3.1, 3.2, 4.4.3, and 4.4.5 - 4.4.7 of the Vienna Report is assumed in much of the following discussion.

The notation defined in the Vienna Report is basically an amalgam of the conditional expressions of LISP [27] with notation from set theory, symbolic logic, and arithmetic. It permits the formal definition of a class of *objects* which may be either *elementary* or *composite*, i.e., composed of one or more other composite or elementary objects. Objects are defined by *predicates* (e.g. the predicates is-symbol and is-prod of section 2.1.1); by convention all predicates are given the prefix "is-". Access to the objects of which a composite object is composed is by means of selector functions, or *selectors* (e.g. the selectors s-symbol and s-defn of section 2.1.1); by convention selectors are given the prefix "s-". To the extent that the *abstract syntax* of a programming language can be identified with a subclass of objects of this type, the notation and concepts of the Vienna Report may be considered a generalization of McCarthy's work on abstract syntax [25,26].

The notation and conventions of the Vienna Report are adopted here verbatim, except for the following changes: Since the concept of list is used frequently in the definitions that follow, it is

convenient to have a concise representation for the length function of lists (cf. section 2.8.2 of the Vienna Report); for this the absolute value symbol of mathematics is adopted, i.e., if is-list(L) then:

$$|L| \underset{Df}{=} \text{length}(L)$$

Also useful with lists is the iterated existential quantifier, defined:

$$\left(\exists_{i=1}^{n} p_i\right) \underset{Df}{=} (\exists p_1)\ldots(\exists p_n)$$

The notation for conditional expressions (cf. section 1.3.1 of the Vienna Report) whose conclusions e_i are all predicates is extended to include expressions of the form:

$$(\ldots,(\exists x)[p_i(x) \rightarrow e_i(x)],\ldots)$$

which is equivalent in meaning to:

$$(\ldots,(\exists x)[p_i(x)] \rightarrow (\exists x)[p_i(x) \wedge e_i(x)],\ldots)$$

Another notation used frequently in the definitions to follow is a quantification over all selectors (cf. 2.1.2 (3) G-Parse Trees). The Greek letters τ and σ (sometimes primed or subscripted) will be used to represent such selectors; i.e., $(\forall\tau)[\ldots]$ is to be understood as an abbreviation for: $(\forall\tau)[\tau\varepsilon S^* \supset (\ldots)]$, where S is the set of all simple selectors used in the sequel (and S* is the set of all simple and composite selectors including the identity selector). Finally, in the definitions given here the symbol " $\wedge$ " will replace

the symbol "&" of the Vienna Report.

The remainder of this section and section 2.2.1 comprise the definition of the objects which represent the abstract form of programs (parse trees) and the abstract form of transformations (pattern trees). For economy of notation the word "tree" will be dropped from the predicates defining these objects, i.e., parse trees will satisfy the predicate is-parse, etc.

To emphasize that parse trees and pattern trees may be adapted for use with any context free phrase structure grammar G, and to clarify which parts of their structure are independent of G, their definitions are given in two parts: first a grammar-independent class of objects is defined (e.g., is-parse); then a *consistency condition* relating this class of objects to a given grammar G is defined (e.g. is-G-parse). Of course, only those elements of the set is-$\hat{\text{p}}$arse which satisfy is-G-parse represent parse trees of a grammar G.

The following subsection introduces the class of elementary objects used in the remaining definitions and defines production rule and grammar in a manner compatible with the notation of the Vienna Report. Parse trees are then defined in section 2.1.2.

2.1.1 Elementary Objects, Productions, and Grammars

The set of *elementary objects, EO,* which will be used to define parse trees and transformations is:

$$EO = \hat{\text{is-symbol}} \cup \hat{\text{is-index}} \cup \{\text{FREE}\} \cup S^*$$

The sets $\hat{\text{is-symbol}}$ and $\hat{\text{is-index}}$ will not be further specified. However, the elements of $\hat{\text{is-symbol}}$ will be related to the symbols of a grammar in 2.1.1 (1) cfpsg. Indices and the element FREE occur only in pattern trees, which are discussed in section 2.2.1, and elements of S^* (compound selectors) occur only in the indefinite skeletons of environment elements, discussed in section 2.3.1.

Before context free grammar can be defined, an object corresponding to a production rule must be introduced. Such an object is called an *abstract production* and satisfies the predicate *is-prod*:

is-prod = (<s-symbol: is-symbol>, <s-defn: is-symbol-list>)

(Recall that, by a convention of section 2.7 of the Vienna Report, "is-symbol-list" denotes a (possibly empty) list of objects satisfying the predicate "is-symbol".) The relation of abstract productions to production rules written in, say, Backus-Naur Form (BNF) will be clarified after definition of context free phrase structure grammars.

Definition 2.1.1 (1) cfpsg

A *context free phrase structure grammar (cfpsg)* G is a four-tuple consisting of three predicates and a symbol:

$$G = (\text{is-G-nonterm}, \text{is-G-term}, \text{is-G-prod}, \xi_s)$$

where:

$$\text{is-G-}\hat{\text{n}}\text{onterm} \cap \text{is-G-}\hat{\text{t}}\text{erm} = \{\} \quad \text{(the empty set)}$$

$$\text{is-G-symbol} = \text{is-G-nonterm} \vee \text{is-G-term}$$

$$\text{is-G-}\hat{\text{s}}\text{ymbol} \subseteq \text{is-}\hat{\text{s}}\text{ymbol}$$

$$\text{is-G-prod}(\pi) \supset \text{is-prod}(\pi) \wedge \text{is-G-nonterm} \circ \text{s-symbol}(\pi)$$

$$\wedge_{\substack{\text{Et} \\ i=1}}^{|\text{s-defn}(\pi)|} \text{is-G-symbol} \circ \text{elem}(i) \circ \text{s-defn}(\pi)$$

$$\text{is-G-nonterm}(\xi_s)$$

The set is-G-$\hat{\text{n}}$onterm is the set of *nonterminal symbols* of G; is-G-$\hat{\text{t}}$erm is the set of *terminal symbols* of G; is-G-$\hat{\text{p}}$rod is the set of *productions* of G; and ξ_s is the *starting symbol* of G. If π is a production of G, the concrete form (in BNF) of π is given (employing the notation introduced in section 2.2.2) by:

$$\text{is-G-prod}(\pi): R[\pi] \Rightarrow \text{rep} \circ \text{s-symbol}(\pi) ::= \underset{i=1}{\overset{|\text{s-defn}(\pi)|}{\text{JUXT}}} \text{rep} \circ \text{elem}(i) \circ \text{s-defn}(\pi)$$

The required relation between productions written in BNF and abstract productions is thus established.

2.1.2 Parse Trees

The parse trees representing derivations (and subderivations) in the grammar G are defined in this section. The general class of parse trees irrespective of a particular grammar G are defined first.

Definition 2.1.2 (1) Parse Trees

A *parse tree* is an object satisfying the predicate *is-parse*:

is-parse = is-final-parse ∨ is-prod-parse
is-final-parse = (<s-symbol: is-symbol>)
is-prod-parse = (<s-symbol: is-symbol>, <s-sub-list: is-parse-list>)

The definition of the consistency condition relating a parse tree p to a specific cfpsg G is facilitated by defining a function immed-prod which extracts what may be called the "immediate production" of p, i.e., the production which permits the derivation of s-sub-list(p) from s-symbol(p). The function immed-prod will later also be applied to pattern trees; therefore its full definition for both parse and pattern trees is given here. (The predicates for pattern trees are defined in section 2.2.1.)

Definition 2.1.2 (2) Immediate Productions

The *immediate production* of a parse tree (or pattern tree) is given by *immed-prod(p)*, where is-parse(p) ∨ is-pat(p):

immed-prod(p) =

(is-prod-parse(p) $\vee$ is-prod-pat(p) $\rightarrow$

μ_0(<s-symbol: s-symbol(p)>,

<s-defn: |s-sub-list(p)| LIST (i=1) s-symbol$\circ$elem(i)$\circ$s-sub-list(p)>),

is-final-parse(p) $\vee$ is-final-pat(p) $\rightarrow$ μ_0(<s-symbol: s-symbol(p)>),

is-indef-pat(p) $\rightarrow$ Ω)

It should be clear that immed-prod satisfies:

is-prod-parse(p) $\vee$ is-prod-pat(p) $\supset$ is-prod(immed-prod(p))

In case p is a final parse (or pattern), immed-prod(p) is just the symbol of p. This convention simplifies certain later definitions.

Now it may happen that if is-parse(p), and G is some cfpsg, is-G-prod(immed-prod(p)). This observation is the basis for the definition of the consistency condition relating parse trees to a particular grammar G. (Recall that $\tau \in S^*$, i.e., τ ranges over all simple and composite selectors, including the identity selector.)

Definition 2.1.2 (3) G-Parse Trees

Let G be a cfpsg. Then a *G-parse-tree* is a parse tree satisfying the consistency condition *is-G-parse(p)*:

is-G-parse(p) = is-parse(p)$\wedge$

$(\forall\tau)$[(is-final-parse$^\circ\tau$(p) $\supset$ is-G-term$\circ$s-symbol$\circ\tau$(p))

$\wedge$(is-prod-parse$\circ\tau$(p) $\supset$ is-G-prod$\circ$immed-prod$\circ\tau$(p))]

This concludes the definition of parse trees. One should note

that if is-G-parse(p) $\wedge$ s-symbol(p) = ξ_s then p is a derivation tree in G in the usual sense. Furthermore, the concrete representation (cf. section 2.2.2) of p is a string in L(G), the language generated by G.

2.2 Syntax of Intra-Grammatical Transformations

In this section the syntax of IGTs will be defined. The definition consists of two main parts: the definition of the abstract form of IGTs, and the definition of their concrete form. The definition of the abstract form is further divided into the definition of G-pattern trees (in a manner analogous to the definition of G-parse trees), the definition of G-IGTs, and the definition of aggregates of IGTs.

2.2.1 Abstract Form of Intra-Grammatical Transformations

Since the tree structures which are the abstract forms of the SDs and SCs of transformations have many features in common, it is convenient to define a single class of tree structures, the pattern trees, of which both sd trees and sc trees are subclasses. This approach clarifies both the similarities and differences between sd trees and sc trees. Note that because the SC of a transformation may contain subtransformation sequences, pattern trees must of necessity contain them; they are defined in definition 2.2.1 (5).

Definition 2.2.1 (1) Pattern Trees

A *pattern tree* is an object satisfying the predicate *is-pat*:

```
is-pat = is-final-pat ∨ is-prod-pat ∨ is-indef-pat
is-final-pat = (<s-symbol: is-symbol>,
                <s-index: is-index ∨ is-Ω>,
                <s-subtrseq: is-subtrseq>)
is-prod-pat = (<s-symbol: is-symbol>,
               <s-index: is-index ∨ is-Ω>,
               <s-sub-list: is-pat-list>,
               <s-subtrseq: is-subtrseq>)
is-indef-pat = (<s-symbol: is-symbol>,
                <s-index: is-index ∨ is-Ω>,
                <s-indef-list: is-pat-or-FREE-nlist>,
                <s-subtrseq: is-subtrseq>)
is-pat-or-FREE = is-pat ∨ is-FREE
```

There are thus three kinds of pattern trees. Final pattern trees and production pattern trees are analogous to final parse trees and

production parse trees; indefinite pattern trees represent indefinites. The suffix "-nlist" used in the definition is the same as the suffix "-list," except that it indicates a non-empty list, i.e., is-indef-pat(t) $\supset$ |s-indef-list(t)| $\geq$ 1. FREE is, of course, the free symbol.

An auxiliary function and predicate will simplify discussions involving pattern trees. (They will also be applied to environment elements, defined in section 2.3.1):

Definition 2.2.1 (2) Variable

The *variable* of a pattern tree (or environment element) is the object composed of its symbol and index (if any), given by *var(t)*, where is-pat(t) $\vee$ is-env-elet(t):

var(t) = μ_0(<s-symbol: s-symbol(t)>, <s-index: s-index(t)>)

Definition 2.2.1 (3) Indexed

A pattern tree (or environment element) is said to be *indexed* (or to have an *indexed symbol*) if it satisfies the predicate *is-indexed(t)*, where is-pat(t) $\vee$ is-env-elet(t):

is-indexed(t) = $\neg$is-$\Omega\circ$s-index(t)

Before the consistency condition for a pattern tree can be stated, it is necessary to introduce some definitions pertaining to subtransformations. Since the consistency conditions for subtransformations are a part of those for pattern trees and transformations, only the

general objects representing subtransformations will be defined now:

Definition 2.2.1 (4) Transformation

A *transformation* is an object satisfying the predicate *is-tr*:

is-tr = (<s-sd: is-pat>, <s-sc: is-pat>)

Definition 2.2.1 (5) Subtransformation Lists and Sequences

A *subtransformation list* is an object satisfying the predicate *is-subtrlist*:

is-subtrlist = is-tr-list

A *subtransformation sequence* is an object satisfying the predicate *is-subtrseq*:

is-subtrseq = is-subtrlist-list

The consistency condition for pattern trees can now be stated. It is rather more complex than that for parse trees and consists of five main parts. (For the definition of immed-prod see 2.1.2 (2) Immediate Production.)

Definition 2.2.1 (6) G-Pattern Trees

Let G be a cfpsg. Then a *G-pattern tree* is a pattern tree satisfying the consistency condition *is-G-pat(t)*:

$$
\begin{aligned}
&\text{is-G-pat}(t) = \text{is-pat}(t) \wedge \\
&\quad (\forall \tau)[(\text{is-final-pat} \circ \tau(t) \supset \text{is-G-symbol} \circ \text{s-symbol} \circ \tau(t)) \\
&\qquad \wedge (\text{is-prod-pat} \circ \tau(t) \supset \text{is-G-prod} \circ \text{immed-prod} \circ \tau(t)) \\
&\qquad \wedge (\text{is-indef-pat} \circ \tau(t) \supset \text{is-G-indef} \circ \tau(t)) \\
&\qquad \wedge (\text{is-tr} \circ \tau(t) \supset \text{is-G-sd} \circ \text{s-sd} \circ \tau(t) \wedge \text{is-G-sc} \circ \text{s-sc} \circ \tau(t) \\
&\qquad\quad \wedge \text{s-symbol} \circ \text{s-sd} \circ \tau(t) = \text{s-symbol} \circ \text{s-sc} \circ \tau(t)) \\
&\qquad \wedge (\text{is-pat} \circ \tau(t) \supset \bigwedge_{i=1}^{|\text{s-subtrseq} \circ \tau(t)|} \bigwedge_{j=1}^{|\text{elem}(i) \circ \text{s-subtrseq} \circ \tau(t)|} \\
&\qquad\quad \text{s-symbol} \circ \text{s-sd} \circ \text{elem}(j) \circ \text{elem}(i) \circ \text{s-subtrseq} \circ \tau(t) = \\
&\qquad\qquad \text{s-symbol} \circ \tau(t))]
\end{aligned}
$$

$$
\begin{aligned}
&\text{is-G-indef}(t) = \text{is-G-nonterm} \circ \text{s-symbol}(t) \\
&\quad |\text{s-indef-list}(t)| - 1 \\
&\qquad \bigwedge_{i=1}^{\text{Et}} \neg (\text{is-FREE} \circ \text{elem}(i) \circ \text{s-indef-list}(t) \\
&\qquad\quad \wedge \text{is-FREE} \circ \text{elem}(i+1) \circ \text{s-indef-list}(t))
\end{aligned}
$$

$$
\begin{aligned}
&\text{is-G-sd}(t) = \text{is-G-pat}(t) \wedge (\forall \tau)[\text{is-pat} \circ \tau(t) \supset \\
&\quad |\text{s-subtrseq} \circ \tau(t)| = 0]
\end{aligned}
$$

$$
\begin{aligned}
&\text{is-G-sc}(t) = \text{is-G-pat}(t) \wedge \\
&\quad (\forall \tau)[(\text{is-indexed} \circ \tau(t) \supset \neg \text{is-prod-pat} \circ \tau(t)) \\
&\qquad \wedge (\neg \text{is-prod-pat} \circ \tau(t) \supset \text{is-indexed} \circ \tau(t) \\
&\qquad\quad \vee \text{is-G-term} \circ \text{s-symbol} \circ \tau(t))]
\end{aligned}
$$

The following aspects of the definition of is-G-pat are noteworthy: Final patterns, unlike final parses, need not have as their symbol a terminal symbol of G. The SD and SC of any transformations occurring in a G-pattern tree (which must, of course, be in a sub-

transformation sequence) must have the same symbol, and the symbols of all SDs of all subtransformations of all subtransformation lists of a subtransformation sequence of a particular pattern subtree must have the same symbol as that pattern subtree. The definition of is-G-indef requires that the symbol of an indefinite be nonterminal, and that no two free symbols occur adjacent to one another in the indefinite list.

The predicate is-G-sd (the *sd condition*) characterizes the subclass of pattern trees (the *sd trees*) which can represent the SD of a transformation as those which do not contain any subtransformations. The predicate is-G-sc (the *sc condition*), which characterizes the *sc trees*, is more complicated. It requires: all production pattern trees must be unindexed; all indefinite pattern trees must be indexed (since the symbol of an indefinite may not be terminal); and all final pattern trees either must be indexed or their symbol must be terminal. These restrictions, together with those placed on transformations, insure that the result of applying a transformation to a G-parse tree is again a G-parse tree.

The consistency condition that a transformation must satisfy in order to be an IGT will now be considered. It must guarantee that when any transformation in the IGT applies, there is an "appropriate" parse tree for every substitution called for by an indexed symbol in the SC of the transformation. Appropriate parse trees can come from three sources: from having been matched in the SD of the transformation, from having been generated (if the symbol is generable),

or from having been matched or generated in a transformation containing the transformation containing the symbol (if the symbol is in a subtransformation). Thus the consistency condition for an IGT must embody the scope rules for subtransformations, which make it quite complex. Some additional nomenclature will simplify the discussion somewhat.

A predicate characterizing the (possibly empty) subset of is-G-symbol which is the set of *generable symbols* is required, and this predicate is *is-generable(ξ)*, where is-G-symbol(ξ). The user of the IGT system is assumed to provide, along with a transformation set sequence, a function *generate(ξ)* and the predicate is-generable, which characterizes the domain of generate. The only constraint placed on generate is that it must satisfy*:

$$\text{is-generable}(\xi) \supset \lambda(p)(\text{is-G-parse}(p) \wedge \text{s-symbol}(p) = \xi)(\text{generate}(\xi))$$

The scope of a transformation can be characterized as follows: Let the *local variables* of a transformation be the variables of all indexed patterns remaining in that transformation after all subtransformation sequences in it have been removed. Then the *scope* of a transformation or subtransformation is the union of the set of the variables of the indexed patterns in its SD and the sets of local variables of all transformations containing it.

---

*The λ-expression is used to state this contraint because generate may be so constructed that successive calls on it with the same argument generate different parse tree, i.e., generate may have side effects.

In these terms, the consistency condition for an IGT can be restated. Every transformation in an IGT satisfies: for every indexed indefinite pattern in the SD of a transformation, there is no other occurrence of a pattern with its variable in an SD in the scope of that transformation; for every indexed nonindefinite pattern in the SC of a transformation, there is a pattern with the corresponding variable in the scope of that transformation, or else the symbol of the pattern is generable; and for every indefinite pattern in the SC of a transformation, there is a suitable indefinite pattern with the corresponding variable in an SD in the scope of that transformation.

The chief problem then, is to define the scope of a transformation. This may be done by considering "factorizations" of composite selectors. A *factorization* of a composite selector $\tau \varepsilon S^*$ is a collection of composite selectors (*factors*) $\sigma_1,\ldots,\sigma_n \varepsilon S^*$ such that $\tau = \sigma_n \circ \ldots \circ \sigma_1$. (Note that composition of selectors is non-commutative, so that the order of the factors is important.)

Consider then a subtransformation t_i contained in the SC of a transformation t, and suppose that $\tau \varepsilon S^*$ selects an indexed nonindefinite pattern in t_i. The requirement that there be a pattern having the corresponding variable within the scope of t_i may be expressed using factorizations of selectors, as:

$$
\begin{aligned}
&(\exists\tau')[\tau' \neq \tau \wedge \text{var}\circ\tau'(t) = \text{var}\circ\tau(t) \\
&\quad \wedge (\exists\sigma)(\exists\sigma_1)(\exists\sigma_2)[\tau = \sigma_1\circ\sigma \wedge \tau' = \sigma_2\circ\sigma \\
&\qquad \wedge\neg(\exists\sigma')(\exists\sigma_1')(\exists\sigma_2')[\sigma_1 = \sigma_1'\circ\sigma' \wedge \sigma_2 = \sigma_2'\circ\sigma' \wedge \sigma' \neq \text{Id}] \\
&\qquad \wedge\neg(\exists\sigma_3)(\exists i)[\sigma_2 = \sigma_3\circ\text{elem}(i)] \\
&\qquad \wedge(\neg(\exists\sigma_3)(\exists\sigma_4)[\sigma_1 = \sigma_4\circ\text{s-subtrseq}\circ\sigma_3] \\
&\qquad\quad \supset (\exists\sigma_3)(\exists\sigma_4)[\tau' = \sigma_4\circ\text{s-sd}\circ\sigma_3] \\
&\qquad\quad\quad \wedge \neg(\exists\sigma_3)(\exists\sigma_4)[\sigma_2 = \sigma_4\circ\text{s-subtrseq}\circ\sigma_3])]]
\end{aligned}
$$

This expression may be made more tractable by introducing some predicates on selectors, two of which will also be useful later.

Definition 2.2.1 (7) Contained and Head

A composite selector σ is *contained* in (is a *head* of) a composite selector τ if σ and τ satisfy *is-contained* (σ,τ), *(is-head*(σ,τ)*)* where $\sigma\in S^*$, $\tau\in S^*$:

$$\text{is-contained}(\sigma,\tau) = (\exists\sigma_1)(\exists\sigma_2)[\tau = \sigma_2\circ\sigma\circ\sigma_1]$$
$$\text{is-head}(\sigma,\tau) = (\exists\sigma_1)[\tau = \sigma_1\circ\sigma]$$

Two other useful predicates are *is-in-sd*(τ) and *is-in-sc*(τ), defined for $\tau\in S^*$:

$$\text{is-in-sd}(\tau) = (\exists\sigma_1)(\exists\sigma_2)[\tau = \sigma_2\circ\text{s-sd}\circ\sigma_1]$$
$$\text{is-in-sc}(\tau) = (\exists\sigma_1)(\exists\sigma_2)[\tau = \sigma_2\circ\text{s-sc}\circ\sigma_1 \wedge \neg\,\text{is-contained}(\text{s-subtrseq},\sigma_2)]$$

Now the predicate *is-in-scope*(τ',τ) determining when a selector τ' is in the scope of a selector τ may be defined:

$$\begin{aligned}
\text{is-in-scope}(\tau',\tau) = \tau' \neq \tau \;\wedge\; &(\exists\sigma)(\exists\sigma_1)(\exists\sigma_2)[\tau=\sigma_1\circ\sigma \wedge \tau' = \sigma_2\circ\sigma \\
&\wedge\neg(\exists\sigma')[\text{is-head}(\sigma',\sigma_1) \wedge \text{is-head}(\sigma',\sigma_2) \wedge \sigma' \neq \text{Id}] \\
&\wedge\neg(\exists i)[\text{is-head}(\text{elem}(i),\sigma_2)] \\
&\wedge(\neg\,\text{is-contained}(\text{s-subtrseq},\sigma_1) \supset \text{is-in-sd}(\tau') \\
&\wedge\neg\,\text{is-contained}(\text{s-subtrseq},\sigma_2))]
\end{aligned}$$

With the predicate is-in-scope the consistency condition defining IGTs can be conveniently stated (cf. definition 2.2.1 (6) for is-G-sd and is-G-sc):

Definition 2.2.1 (8) IGT

Let G be a cfpsg. An *intra-grammatical transformation (IGT)* over G is an object satisfying the predicate *is-G-IGT(t)*:

$$\begin{aligned}
\text{is-G-IGT}(t) = \;&\text{is-tr}(t) \wedge \text{is-G-sd}\circ\text{s-sd}(t) \wedge \text{is-G-sc}\circ\text{sc}(t) \\
&\wedge \text{s-symbol}\circ\text{s-sd}(t) = \text{s-symbol}\circ\text{s-sc}(t) \\
&\wedge(\forall\tau)[(\text{is-in-sd}\circ\tau(t) \wedge \text{is-indef-pat}\circ\tau(t) \wedge \text{is-indexed}\circ\tau(t) \\
&\qquad\supset \neg(\exists\tau')[\text{is-in-sd}\circ\tau'(t) \wedge \text{is-corresp-pat}(\tau',\tau,t)]) \\
&\qquad\wedge(\text{is-in-sc}\circ\tau(t) \supset [\text{is-final-pat}\circ\tau(t) \wedge \text{is-indexed}\circ\tau(t) \\
&\qquad\qquad\supset(\exists\tau')[\text{is-corresp-pat}(\tau',\tau,t)] \\
&\qquad\qquad\qquad\vee\,\text{is-generable}\circ\text{s-symbol}\circ\tau(t)] \\
&\qquad\wedge[\text{is-indef-pat}\circ\tau(t) \\
&\qquad\qquad\supset(\exists\tau')[\text{is-corresp-indef}(\tau',\tau,t)]])]
\end{aligned}$$

$$\text{is-corresp-pat}(\tau',\tau,t) = (\text{var}\circ\tau'(t) = \text{var}\circ\tau(t) \wedge \text{is-in-scope}(\tau',\tau))$$

$$\begin{aligned}
\text{is-corresp-indef}(\tau',\tau,t) = \;&\text{is-corresp-pat}(\tau',\tau,t) \wedge \text{is-in-sd}(\tau') \\
&\wedge \text{is-indef-pat}\circ\tau'(t) \wedge |\text{s-indef-list}\circ\tau'(t)| = |\text{s-indef-list}\circ\tau(t)| \\
&\wedge\left(\underset{i=1}{\overset{|\text{s-indef-list}\circ\tau(t)|}{\text{Et}}} \text{s-symbol}\circ\text{elem}(i)\circ\text{s-indef-list}\circ\tau'(t)\right. \\
&\qquad\left.= \text{s-symbol}\circ\text{elem}(i)\circ\text{s-indef-list}\circ\tau(t)\right)
\end{aligned}$$

Note that the predicate is-corresp-indef requires that, for an indefinite pattern in an sc tree, there be a pattern with the corresponding variable in an SD, and it must also be indefinite. Furthermore, the sequence of symbols in the indefinite lists of both patterns must be the same.*

The definition of the various aggregates of transformations completes the specification of the abstract form of IGTs.

Definition 2.2.1 (9) Transformation Lists

Let G be a cfpsg. A *transformation list* is an object satisfying *is-G-trlist(t):*

$$\text{is-G-trlist}(t) = \text{is-G-IGT-list}(t) \wedge [|t| > 0 \supset \underset{i=1}{\overset{|t|-1}{\text{Et}}} \text{ s-symbol}\circ\text{s-sd}\circ\text{elem}(i)(t) = \text{s-symbol}\circ\text{s-sd}\circ\text{elem}(i+1)(t)]$$

If s-symbol∘s-sd∘elem(1)(t) = ξ, t is sometimes called a *ξ-transformation list.*

Definition 2.2.1 (10) Transformation Sets

Let G be a cfpsg. A *transformation set* is an object satisfying *is-G-trset(S):*

---

*Actually, in this condition s-symbol∘FREE = Ω; however, since the lengths of the two indefinite lists must be the same and s-symbol is defined for all other objects which can occur in the list,

$$\text{s-symbol}\circ\text{elem}(i)\circ\text{s-indef-list}\circ\tau(t) = \Omega \equiv \text{is-FREE}\circ\text{elem}(i)\circ\text{s-indef-list}\circ\tau(t)$$

$$\text{is-G-trset}(S) = (\forall \ell)[\ell \varepsilon S \supset \text{is-G-trlist}(\ell) \wedge |\ell| > 0 \wedge \neg(\exists \ell')[\ell' \varepsilon S \wedge \text{s-symbol} \circ \text{s-sd} \circ \text{elem}(1)(\ell') = \text{s-symbol} \circ \text{s-sd} \circ \text{elem}(1)(\ell)]]$$

Definition 2.2.1 (11) Transformation Set Sequences

Let G be a cfpsg. A *transformation set sequence* is an object satisfying *is-G-trsetseq:*

$$\text{is-G-trsetseq} = \text{is-G-trset-list}$$

The definition of the abstract form of IGTs is thus complete. The following section defines a concrete representation for IGTs based on this abstract form.

2.2.2 Concrete Form of Intra-Grammatical Transformations

The abstract forms of parse trees and IGTs, while necessary for the definition of the semantics of transformations, are rather cumbersome and not well adapted to writing programs and transformations. Therefore concrete forms, which are the forms in which programs and transformations are normally written and displayed (cf. section 1.1), are introduced. These concrete forms are defined by a representation function which interprets abstract forms to produce corresponding concrete forms.

The concept of representation system and the notation of section 3.2 of the Vienna Report provide the basis for constructing the representation function. The *representation system* developed here is the 5-tuple: <A,T,{R},R,R> where: A is the abstract syntax described in sections 2.1.1, 2.1.2, and 2.2.1; T is the set of *represented symbols**:

$$T = \text{rep}(\text{is-}\hat{\text{symbol}} \cup \text{is-}\hat{\text{index}}) \cup \{\text{'SD'},\text{'SC'},\text{==>},\{,\},\text{'IND'},\text{''},?\}$$

(The function rep will not be specified except to say that it is a 1-1 mapping between symbols and indices and their written representations); R is the single nonterminal name of the representation system (*the representation function*); and R is the conditional replacement schema defining R, given below.

---

*In the computer implementation of IGTs, "{" is represented by "@", and "}" by "%" (cf. sections 3.1.2 and 3.2.2).

Since lists occur frequently in the abstract syntax of parse trees and IGTs, it is worthwhile to introduce an abbreviated notation for the representation of the elements of a list; this is the juxtaposition operator JUXT, defined when is-list(L). (λ is the empty symbol.)

$$\underset{i=1}{\overset{|L|}{\text{JUXT}}}\ \mathcal{R}[\text{elem}(i)(L)] \underset{\text{DF}}{=} (|L| = 0 \rightarrow \lambda,\ |L| > 0 \rightarrow \mathcal{R}[\text{elem}(1)(L)]\ldots\mathcal{R}[\text{elem}(|L|)(L)])$$

Definition 2.2.2 (1) Representation Function

The representation function $\mathcal{R}$ is defined for composite objects by the conditional replacement schema R:

$$\text{is-G-trsetseq}(t) \vee \text{is-subtrseq}(t): \mathcal{R}[t] \Rightarrow \underset{i=1}{\overset{|t|}{\text{JUXT}}}\ \mathcal{R}[\text{elem}(i)(t)]$$

$$\text{is-G-trset}(t) \wedge \left(\underset{i=1}{\overset{n}{\text{Et}}}\ \text{is-G-trlist}(\ell_i)\right) \wedge t = \{\ell_1,\ldots,\ell_n\} \wedge \ell = \underset{i=1}{\overset{n}{\text{LIST}}}\ \ell_i:$$

$$\mathcal{R}[t] = \left\{\ \underset{i=1}{\overset{|\ell|}{\text{JUXT}}}\ \mathcal{R}[\text{elem}(i)(\ell)]\ \right\}$$

$$\text{is-G-trlist}(t) \wedge |t| > 0:$$

$$\mathcal{R}[t] \Rightarrow \text{rep}\circ\text{s-symbol}\circ\text{s-sd}\circ\text{elem}(1)(t)\ \left\{\ \underset{i=1}{\overset{|t|}{\text{JUXT}}}\ \mathcal{R}[\text{elem}(i)(t)]\ \right\}$$

$$\text{is-subtrlist}(t): \mathcal{R}[t] \Rightarrow \left\{\ \underset{i=1}{\overset{|t|}{\text{JUXT}}}\ \mathcal{R}[\text{elem}(i)(t)]\ \right\}$$

$$\text{is-tr}(t): \mathcal{R}[t] \Rightarrow \text{'SD'}\ \ \mathcal{R}[\text{s-sd}(t)]\ \Longrightarrow\ \mathcal{R}[\text{s-sc}(t)]\ \ \text{'SC'}$$

$$\text{is-final-pat}(t) \vee \text{is-final-parse}(t):$$

$$\mathcal{R}[t] \Rightarrow \text{rep}\circ\text{s-symbol}(t)\ \mathcal{R}[\text{s-index}(t)]\ \mathcal{R}[\text{s-subtrseq}(t)]$$

$$\text{is-prod-pat}(t): \mathcal{R}[t] \Rightarrow \text{rep}\circ\text{s-symbol}(t)\ \mathcal{R}[\text{s-index}(t)]$$

$$\left\{\ \underset{i=1}{\overset{|\text{s-sub-list}(t)|}{\text{JUXT}}}\ \mathcal{R}[\text{elem}(i)\circ\text{s-sub-list}(t)]\ \right\}\ \mathcal{R}[\text{s-subtrseq}(t)]$$

```
is-prod-pat(t) ∧ ¬(is-indexed(t) ∨ |s-subtrseq(t)| > 0) ∨ is-prod-parse(t):
                    |s-sub-list(t)|
        R[t] =>          JUXT        R[elem(i)∘s-sub-list(t)]
                         i=1

is-indef-pat(t): R[t] => rep∘s-symbol(t) R[s-index(t)]
                    |s-indef-list(t)|
        { 'IND'          JUXT        R[elem(i)∘s-indef-list(t)] }
                         i=1
        R[s-subtrseq(t)]

is-indef-pat(t) ∧ (∃i)[is-FREE∘elem(i)∘s-indef-list(t)]:
        R[t] => rep∘s-symbol(t) R[s-index(t)]
           |s-indef-list(t)|
        {        JUXT        R[elem(i)∘s-indef-list(t)] } R[s-subtrseq(t)]
                 i=1

is-index(t): R[t] => " rep(t) "

is-FREE(t): R[t] => ?

is-Ω: R[t] => λ
```

One should note two points about the representation function R: (1) The abstract forms of certain IGTs have more than one concrete form; when this is true, they are all equivalent. (2) When R is applied to a ξ_s-parse tree of a cfpsg G, the result is a string in the language generated by G:

$$\text{is-G-parse}(t) \wedge \text{s-symbol}(t) = \xi_s \supset R[t] \in L(G)$$

2.3 Semantics of Intra-Grammatical Transformations

The semantic definition of IGTs comprises two principal parts: the specification of the result of applying a transformation to a parse tree, and the specification of the sequencing of aggregates of transformations and subtransformations. Since the application of a transformation necessarily involves applying the subtransformation sequences in it, these two specifications are mutually dependent.

In addition to defining these two aspects of the semantics of IGTs, this section also introduces the concept of an *environment*. Basically, the environment of an IGT is the information which, in addition to that in the SD and SC of the IGT, relates the transformed parse tree to the parse tree to which the transformation is applied. In an algorithmic implementation of IGTs, the environment is constructed during matching of the SD of the IGT against the parse tree and is used to look up the values of variables in the SC during construction of the transformed parse tree. However, in the following definition of the semantics of IGTs, no attempt is made to specify the computation of an environment, but rather the existence of an environment relating two parse trees and a transformation is taken as the condition for that transformation to apply to one parse tree to produce the other. Since for indefinites more than one suitable environment may exist, this section concludes with a discussion of an order relation for environments, which permits the selection of a unique environment (the minimal, or left-most one) for each application of a transformation to a parse tree.

2.3.1 Environment of an Intra-Grammatical Transformation

An environment of an IGT is a set of environment elements, each of which consists of a variable (in the sense of definition 2.2.1 (2)), a parse tree (the value of the element), and possibly an *indefinite skeleton*. The indefinite skeleton is a list of selectors which indicate the nodes in the parse tree matched by the non-FREE elements of the indefinite list of a corresponding indefinite pattern. In this way the tree structure represented by the FREE symbols of an indefinite pattern is assured to be the same for the same variable in the SD and SC.

Following the pattern of previous definitions, the objects representing environments are first defined, and then certain consistency conditions, which primarily define the semantics of FREE symbols, are placed on them.

Definition 2.3.1 (1) Environments and Environment Elements

An *environment* is a set satisfying the predicate *is-env*; an environment element is an object satisfying the predicate *is-env-elet*:

is-env(E) = $(\forall e)(e \in E \supset$ is-env-elet(e))

is-env-elet = (<s-symbol: is-symbol>,
<s-index: is-index $\vee$ is-Ω>, <s-parse: is-parse>,
<s-indef-skel: is-sel-or-s-FREE-nlist $\vee$ is-Ω>)

is-sel-or-s-FREE = is-sel $\vee$ is-s-FREE

$\hat{\text{is-sel}} = S^*$

is-s-FREE(s) = $(\forall o)$[is-FREE$\circ$s(o)]

Note that s-FREE is a constant selector, having as value the elementary object FREE regardless of the object to which it is applied. (By a convention of the Vienna Report (section 2.3) s-FREE is not an element of S^*.)

The consistency condition for environment elements is rather complex, inasmuch as it helps to define the semantics of indefinites (cf. section 1.1.3). The condition must describe the following relations: the subtree matched by a non-FREE element left of another in the indefinite list must be left of the subtree matched by the other; a non-FREE element at the left (right) end of an indefinite list must match a leftmost (rightmost) subtree; and adjacent non-FREE elements must match adjacent subtrees (in the sense that the terminal strings of their subtrees are adjacent). Predicates characterizing these relationships can be defined in terms of the selectors accessing the subtrees (and their factorizations).

The subtree selected by τ_1 is *left of* the subtree selected by τ_2 if they satisfy the predicate *is-left-of*(τ_1,τ_2) where $\tau_1 \in S^* \wedge \tau_2 \in S^*$:

$$\text{is-left-of}(\tau_1,\tau_2) = (\exists\sigma)(\exists\sigma_1)(\exists\sigma_2)(\exists i_1)(\exists i_2)$$
$$[\tau_1 = \sigma_1 \circ \text{elem}(i_1) \circ \sigma \wedge \tau_2 = \sigma_2 \circ \text{elem}(i_2) \circ \sigma \wedge i_1 < i_2]$$

The subtree selected by τ in p is a *leftmost* subtree of p if they satisfy the predicate *is-leftmost* (τ,p) where $\tau \in S^* \wedge$ is-G-parse(p):

$$\text{is-leftmost}(\tau,p) = (\forall\sigma)(\forall\sigma')(\forall i)[\tau = \sigma' \circ \text{elem}(i) \circ \sigma \supset i=1]$$

The subtree selected by τ in p is a *rightmost* subtree of p if the predicate *is-rightmost*(τ,p) is satisfied, where $\tau \in S^* \wedge$ is-G-parse(p):

$$\text{is-rightmost}(\tau,p) = (\forall\sigma)(\forall\sigma')(\forall i)[\tau = \sigma'\circ\text{elem}(i)\circ\sigma \supset i = |\sigma(p)|]$$

(Note that the argument p is not used in is-leftmost; this is an artifact of the notation for lists.) The subtree selected by τ_1 is *adjacent* to that selected by τ_2 in p if the predicate *is-adjacent*(τ_1,τ_2,p) is satisfied, where $\tau_1 \in S^* \wedge \tau_2 \in S^* \wedge$ is-G-parse(p):

$$\begin{aligned}\text{is-adjacent}(\tau_1,\tau_2,p) = {} & (\exists\sigma)(\exists\sigma_1)(\exists\sigma_2)(\exists i) \\ & [\tau_1 = \sigma_1\circ\text{elem}(i)\circ\sigma \wedge \tau_2 = \sigma_2\circ\text{elem}(i+1)\circ\sigma \\ & \wedge \text{is-rightmost}(\sigma_1, \text{elem}(i)\circ\sigma(p)) \\ & \wedge \text{is-leftmost}(\sigma_2, \text{elem}(i+1)\circ\sigma(p))]\end{aligned}$$

With these definitions, the consistency condition for environment elements becomes:

Definition 2.3.1 (2) G-Environment Elements

A *G-environment element* is an environment element satisfying the consistency condition *is-G-env-elet:*

$$\begin{aligned}
&\text{is-G-env-elet}(e) = \text{is-env-elet}(e) \wedge \text{is-G-parse}\circ\text{s-parse}(e) \\
&\quad \wedge \text{s-symbol}(e) = \text{s-symbol}\circ\text{s-parse}(e) \wedge [\neg\text{is-}\Omega\circ\text{s-indef-skel}(e) \\
&\qquad |\text{s-indef-skel}(e)| \\
&\qquad \supset \quad \underset{i=1}{\overset{}{\text{Et}}} \quad (\neg\text{is-s-FREE}\circ\text{elem}(i)\circ\text{s-indef-skel}(e) \supset \\
&\qquad\quad (\exists\sigma_i)[\sigma_i = \text{elem}(i)\circ\text{s-indef-skel}(e) \\
&\qquad\qquad \wedge \text{is-parse}\circ\sigma_i\circ\text{s-parse}(e) \\
&\qquad\qquad \wedge (i=1 \supset \text{is-leftmost}\ (\sigma_i, \text{s-parse}(e))) \\
&\qquad\qquad \wedge (i = |\text{s-indef-skel}(e)| \supset \text{is-rightmost}(\sigma_i, \text{s-parse}(e))) \\
&\qquad\qquad \wedge \underset{j=1}{\overset{i-1}{\text{Et}}} (\neg\ \text{is-s-FREE}\circ\text{elem}(j)\circ\text{s-indef-skel}(e) \\
&\qquad\qquad\quad \supset [\text{is-left-of}(\text{elem}(j)\circ\text{s-indef-skel}(e), \sigma_i) \\
&\qquad\qquad\qquad \wedge (j=i-1 \supset \text{is-adjacent}(\text{elem}(j)\circ\text{s-indef-skel}(e), \\
&\qquad\qquad\qquad\quad \sigma_i, \text{s-parse}(e)))])])]
\end{aligned}$$

Finally, a G-environment is an environment in which no two indexed elements have the same variable:

Definition 2.3.1 (3) G-Environment

A *G-environment* is an environment satisfying the consistency condition *is-G-env*:

$$\begin{aligned}
&\text{is-G-env}(E) = \text{is-env}(E) \wedge (\forall e)[e \in E \supset \text{is-G-env-elet}(e) \\
&\quad \wedge(\text{is-indexed}(e) \supset \neg(\exists e')[e' \in E \wedge \text{var}(e') = \text{var}(e)])]
\end{aligned}$$

The introduction of two additional predicates relating environment elements to parse trees and indefinite patterns, respectively, complete the discussion of environments:

Definition 2.3.1 (4) Instance

A parse tree p is an *instance* of an environment element e if the predicate *is-instance(p,e,t)* is satisfied, where is-G-parse(p) $\wedge$ is-G-env-elet(e) $\wedge$ is-pat(t) (cf. definition 2.2.1 (7) for is-head):

$$\begin{aligned}\text{is-instance}(p,e,t) = (\forall\tau)[&\text{is-parse}\circ\tau(p) \wedge (\neg\,\text{is-indef-pat}(t) \vee \\ &\neg(\exists i)[\text{is-head}(\text{elem}(i)\circ\text{s-indef-skel}(e),\tau)]) \\ &\supset \text{immed-prod}\circ\tau(p) = \text{immed-prod}\circ\tau\circ\text{s-parse}(e)]\end{aligned}$$

Basically, a parse tree p is an instance of an environment element e if p = s-parse(e) except, if t is indefinite, for the subtrees selected by non-FREE elements of s-indef-skel(e). Note that for nonindefinite patterns, the definition reduces to p = s-parse(e).

Definition 2.3.1 (5) Compatible

An environment element is *compatible* with an indefinite pattern if the predicate *is-compat(e,t)* is satisfied, where is-G-env-elet(e)$\wedge$ is-indef-pat(t):

$$\begin{aligned}\text{is-compat}(e,t) = [&\text{var}(e) = \text{var}(t) \wedge \neg\,\text{is-}\Omega\circ\text{s-indef-skel}(e) \\ &\wedge |\text{s-indef-skel}(e)| = |\text{s-indef-list}(t)| \\ &\wedge \underset{i=1}{\overset{|\text{s-indef-list}(t)|}{Et}} (\exists\sigma_i)[\sigma_i = \text{elem}(i)\circ\text{s-indef-skel}(e) \\ &\quad \wedge \text{s-symbol}\circ\sigma_i\circ\text{s-parse}(e) \\ &\quad = \text{s-symbol}\circ\text{elem}(i)\circ\text{s-indef-list}(t)]]\end{aligned}$$

Thus an environment element corresponding to an indefinite pattern is compatible with it provided the element has an indefinite skeleton with FREE elements in the same positions as the FREE symbols in the indefinite list, and whose non-FREE elements select subtrees whose symbols are the same as those of their corresponding subpatterns in the indefinite list.

2.3.2 Predicates Defining the Semantics of Transformations

The predicate which defines the result of applying a single IGT to a parse tree necessarily uses predicates defining the semantics of subtransformation sequences (which may be applied to subtrees of the transformed parse tree) and also those defining the semantics of transformation sets (which are reapplied to the transformed parse tree in bottom-up order). Therefore, these predicates, which define the sequencing rules discussed in sections 1.1.2 and 1.1.4, will be defined first. One may keep in mind that the predicate is-tr-result (p_2,p_1,t,E,S) (definition 2.3.2 (8)) is true if p_2 is the result of recursively reapplying the transformation set S to the result of applying the transformation t to p_1, with the environment E.

The first predicate to be defined is the one which characterizes the result of applying a transformation list. It employs the predicate is-indef-min (defined in section 2.3.3) to establish the uniqueness of the environment (with respect to its indefinite elements) used by is-tr-result. For the purpose of understanding the definitions of this section, is-indef-min may be taken to be true for any environment E. (Note that conditional expressions of the type discussed in section 2.1 are used in this and several of the following definitions.)

Definition 2.3.2 (1) Trlist Result

A parse tree p_2 is the *result of applying a transformation list* to a parse tree p_1 if the predicate *is-trlist-result* (p_2,p_1,ℓ,S) is satisfied, where is-G-parse(p_2) $\wedge$ is-G-parse(p_1) $\wedge$ is-G-trlist(ℓ)$\wedge$ is-G-trset(S):

$$
\begin{aligned}
&\text{is-trlist-result}(p_2,p_1,\ell,S) = \\
&\quad (|\ell| = 0 \rightarrow p_2 = p_1, \\
&\quad (\exists p)[(\exists E)[\text{is-G-parse}(p) \wedge \text{is-G-env}(E) \\
&\qquad \wedge \text{is-indef-min}(E,p_1,\text{head}(\ell),\{\}) \\
&\qquad \wedge \text{is-tr-result}(p,p_1,\text{head}(\ell),E,S)] \rightarrow p_2 = p], \\
&\quad T \rightarrow \text{is-trlist-result}(p_2,p_1,\text{tail}(\ell),S))
\end{aligned}
$$

Basically, is-trlist-result is true if p_2 is the result of applying the *first* transformation in the transformation list that applies to p_1; or, if none applies, if $p_2 = p_1$.

The next two predicates define the result of applying a transformation set to a parse tree. The first characterizes the result of applying a transformation set to the first level of the parse tree, while the second defines the result of applying the set throughout the tree in bottom-up, left-to-right order (cf. section 1.1.2).

Definition 2.3.2 (2) Trset Result

A parse tree p_2 is the *result of applying a transformation set* to a parse tree p_1 if the predicate *is-trset-result*(p_2,p_1,S) is satisfied, where is-G-parse(p_2) $\wedge$ is-G-parse(p_1) $\wedge$ is-G-trset(S):

$$
\begin{aligned}
&\text{is-trset-result}(p_2,p_1,S) = \\
&\quad ((\exists \ell)[\ell \in S \wedge \text{s-symbol}\circ\text{s-sd}\circ\text{elem}(1)(\ell) = \text{s-symbol}(p_1) \\
&\qquad \rightarrow \text{is-trlist-result}(p_2,p_1,\ell,S)], \\
&\quad T \rightarrow p_2 = p_1)
\end{aligned}
$$

Definition 2.3.2 (3) Trset Recursive Result

A parse tree p_2 is the *result of the recursive application of a transformation set* throughout a parse tree p_1 if the predicate *is-trset-rec-result*(p_2,p_1,S) is satisfied, where is-G-parse$(p_2) \wedge$ is-G-parse$(p_1) \wedge$ is-G-trset(S):

$$
\begin{aligned}
&\text{is-trset-rec-result}(p_2,p_1,S) = \\
&\quad (\text{is-final-parse}(p_1) \supset \text{is-trset-result}(p_2,p_1,S)) \\
&\quad \wedge(\text{is-prod-parse}(p_1) \supset (\exists p)[\text{is-G-parse}(p) \\
&\qquad \wedge |\text{s-sub-list}(p)| = |\text{s-sub-list}(p_1)| \\
&\qquad \wedge \left[\begin{array}{l} |\text{s-sub-list}(p_1)| \\ \underset{i=1}{\overset{}{\text{Et}}} \end{array} \text{is-trset-rec-result}(\text{elem}(i)\circ\text{s-sub-list}(p), \right. \\
&\qquad\qquad \left. \text{elem}(i)\circ\text{s-sub-list}(p_1),S)\right] \\
&\qquad \wedge \text{is-trset-result}(p_2,p,S)])
\end{aligned}
$$

The result of applying a transformation set sequence is defined in terms of the recursive application of each transformation set in the sequence to the result of its predecessor:

Definition 2.3.2 (4) Trsetseq Result

A parse tree p is the *result of applying a transformation set sequence* to a parse tree p_0 if the predicate *is-trsetseq-result*(p,p_0,q) is satisfied, where is-G-parse$(p) \wedge$ is-G-parse$(p_0) \wedge$ is-G-trsetseq(q):

$$\text{is-trsetseq-result}(p,p_0,q) = \left(\exists_{i=1}^{|q|} p_i\right)\left[\left(\text{Et}_{i=1}^{|q|} [\text{is-G-parse}(p_i) \wedge \text{is-trset-rec-result}(p_i,p_{i-1},\text{elem}(i)(q))]\right) \wedge p = p_{|q|}\right]$$

Finally, the function which returns the result of applying a transformation set sequence to a parse tree is defined as follows:

Definition 2.3.2 (5) Transform

The *result of applying a transformation set sequence to a parse tree* is the value of a function *transform(p,q)*, where is-G-parse(p)∧ is-G-trsetseq(q), which satisfies:

is-trsetseq-result(transform(p,q),p,q)

Note that transform (p,q) always has a value, since if no transformation in the sequence ever applies, is-trsetseq-result(p,p,q).

The two predicates which define sequencing for subtransformations are quite similar to is-trlist-result and is-trsetseq-result. However, the Markov sequencing must be explicitly built into is-subtrlist-result, since it does not arise implicitly from recursive reapplication as it does for transformation lists. Moreover, is-subtrlist-result must also pass to the subtransformations the environment of the transformation in which they occur, augmented by an environment containing elements corresponding to variables occurring in the subtransformation but not in the transformations containing it.

Definition 2.3.2 (6) Subtrlist Result

A parse tree p_2 is the *result of applying a subtransformation list* to a parse tree p_1 if the predicate *is-subtrlist-result* (p_2,p_1,ℓ,E,ℓ',S) is satisfied, where is-G-parse(p_2) $\wedge$ is-G-parse(p_1) $\wedge$ is-subtrlist(ℓ) $\wedge$ is-G-env(E) $\wedge$ is-subtrlist(ℓ') $\wedge$ is-G-trset(S):

$$\begin{aligned}
&\text{is-subtrlist-result}(p_2,p_1,\ell,E,\ell',S) = \\
&\quad (|\ell| = 0 \rightarrow p_2 = p_1, \\
&\quad (\exists p)[(\exists E')[\text{is-G-parse}(p) \wedge \text{is-G-env}(E' \cup E) \\
&\quad \wedge \text{is-indef-min}(E',p_1,\text{head}(\ell),E) \\
&\quad \wedge \text{is-tr-result}(p,p_1,\text{head}(\ell),E' \cup E,S)] \\
&\qquad \rightarrow \text{is-subtrlist-result}(p_2,p,\ell',E,\ell',S)], \\
&\quad T \rightarrow \text{is-subtrlist-result}(p_2,p_1,\text{tail}(\ell),E,\ell',S))
\end{aligned}$$

Definition 2.3.2 (7) Subtrseq Result

A parse tree p is the *result of applying a subtransformation sequence* to a parse tree p_0 if the predicate *is-subtrseq-result* (p,p_0,q,E,S) is satisfied, where is-G-parse(p) $\wedge$ is-G-parse(p_0) $\wedge$ is-subtrseq(q) $\wedge$ is-G-env(E) $\wedge$ is-G-trset(S):

$$\begin{aligned}
&\text{is-subtrseq-result}(p,p_0,q,E,S) = \left(\exists_{i=1}^{|q|} p_i\right)\left[\left(\text{Et}_{i=1}^{|q|} [\text{is-G-parse}(p_i)\right.\right. \\
&\quad \wedge \text{is-subtrlist-result}(p_i,p_{i-1},\text{elem}(i)(q),E,\text{elem}(i)(q),S)]) \\
&\quad \left.\left. \wedge p = p_{|q|}\right]\right.
\end{aligned}$$

The definition of the sequencing of transformations and subtransformations is now complete, and only the definition of the semantics of an individual transformation, via the predicate is-tr-result, remains. This predicate is defined in terms of one further predicate,

is-match; is-tr-result requires that the parse tree being transformed match the SD of the transformation and that the transformed parse tree match the SC of the transformation.

Definition 2.3.2 (8) Tr Result

A parse tree p_2 is the *result of applying an IGT* to a parse tree p_1 if the predicate *is-tr-result*(p_2,p_1,t,E,S) is satisfied, where is-G-parse(p_2) $\wedge$ is-G-parse(p_1) $\wedge$ is-G-IGT(t) $\wedge$ is-G-env(E) $\wedge$ is-G-trset(S):

$$\text{is-tr-result}(p_2,p_1,t,E,S) = (\exists p)[\text{is-G-parse}(p) \wedge \text{is-match}(p_1,\text{s-sd}(t),E,S) \wedge \text{is-match}(p,\text{s-sc}(t),E,S) \wedge \text{is-trset-result}(p_2,p,S)]$$

Definition 2.3.2 (9) Match

An SD or SC of a transformation *matches* a parse tree if the recursively defined predicate *is-match*(p,t,E,S) is satisfied, where is-G-parse(p) $\wedge$ (is-G-sd(t) $\vee$ is-G-sc(t)) $\wedge$ is-G-env(E) $\wedge$ is-G-trset(S):

$$\begin{aligned}\text{is-match}(p,t,E,S) = {} & (\exists p'')(\exists p')(\exists e)\\ & [\text{is-G-parse}(p'') \wedge \text{is-G-parse}(p')\\ & \wedge \text{s-symbol}(p) = \text{s-symbol}(p'') = \text{s-symbol}(p') = \text{s-symbol}(t)\\ & \wedge [\text{is-indexed}(t) \vee \text{is-indef-pat}(t) \supset e \varepsilon E\\ & \quad \wedge \text{var}(e) = \text{var}(t) \wedge \text{is-instance}(p',e,t)]\\ & \wedge [\text{is-prod-pat}(t) \supset |\text{s-sub-list}(p')| = |\text{s-sub-list}(t)|\\ & \quad |\text{s-sub-list}(t)|\\ & \wedge \quad \underset{i=1}{\overset{t}{E}} \quad \text{is-match}(\text{elem}(i)\circ\text{s-sub-list}(p'),\\ & \quad \text{elem}(i)\circ\text{s-sub-list}(t),E,S)]\end{aligned}$$

$$\wedge\,[\text{is-indef-pat}(t) \supset \text{is-compat}(e,t)$$
$$\bigwedge_{i=1}^{|\text{s-indef-list}(t)|} (\neg\,\text{is-FREE}\circ\text{elem}(i)\circ\text{s-indef-list}(t)$$
$$\supset (\exists\sigma_i)[\sigma_i = \text{elem}(i)\circ\text{s-indef-skel}(e)$$
$$\wedge\,\text{is-match}(\sigma_i(p'), \text{elem}(i)\circ\text{s-indef-list}(t), E, S)])]$$
$$\wedge[\text{is-prod-parse}(p') \supset |\text{s-sub-list}(p'')| = |\text{s-sub-list}(p')|$$
$$\bigwedge_{i=1}^{|\text{s-sub-list}(p')|} \text{is-trset-rec-result}$$
$$(\text{elem}(i)\circ\text{s-sub-list}(p''), \text{elem}(i)\circ\text{s-sub-list}(p'), S)]$$
$$\wedge\text{is-subtrseq-result}(p,p'',\text{s-subtrseq}(t),E,S)]$$

This rather lengthy recursive definition may be understood as follows: There must exist two parse trees satisfying some obvious restrictions, which act as "intermediate results." If the pattern is indexed or indefinite there must also exist an environment element corresponding to it, (var(e) = var(t)), and the parse tree p' must be an instance of it. (Recall that for non-indefinite patterns, is-instance (definition 2.3.1 (4)) represents equality.)

The recursive phase of the definition determines that the subtrees of the parse tree p' match the subtrees of the pattern tree. (But if the pattern is final no further restrictions are placed on the remainder of p'.) For a production pattern the recursion is on the elements of the sub-lists of p' and t; for an indefinite pattern, e must be compatible with t (cf. definition 2.3.1 (5)), and the recursion is on the non-FREE elements of the indefinite list of t, descending in p' all the way to the subtrees selected by the corresponding elements of the indefinite skeleton of e.

The next phase of the definition determines that the subtrees (if any) of p" are the result of applying the transformation set S recursively to the respective subtrees of p'. The final phase then consists of checking that p is the result of applying the subtransformation sequence of t to p". (If there is no subtransformation sequence of t, this becomes p = p".)

The following points should be noted about this definition. First, when is-match is applied to the SD of a transformation, the last two phases are vacuous. This is obvious for the subtransformation sequence phase, since the consistency condition is-G-sd permits no subtransformation sequences in the SD. That it is also true for the transformation set phase is a consequence primarily of the definitions of is-trset-rec-result and is-match: together they insure that before any transformation is applied at a given point in a parse tree, all its subtrees have been fully transformed by the transformation set. This redundancy in the definition is tolerated to retain the symmetry of employing a single match function for both the SD and SC of a transformation.

Second, there is also some redundancy in the transformation set phase when is-match is applied to the SC of a transformation. That is, for non-indefinite patterns one may employ is-trset-result in place of is-trset-rec-result. This is clear for production patterns, since is-trset-result will have been applied to the subtrees as a consequence of the recursion of is-match. For final patterns, it is a consequence of the definition of is-G-sc, since a non-final parse

tree corresponding to a final pattern tree must appear also in the parse tree matched by the SD (or else be generable), hence it must have already been fully transformed (assuming generated parse trees are fully transformed). Even for indefinite patterns it is not necessary to perform a full recursive reapplication of the transformation set, since it can only reapply at a node whose selector is a head of one of the selectors in the corresponding indefinite skeleton (again because the parse tree of the corresponding environment element must have appeared in the parse tree matched by the SD). This slight redundancy is tolerated to maintain the simplicity of the definition.

This completes the definition of the semantics of IGTs except for the question of the uniqueness of environments, discussed in the following section.

2.3.3 Uniqueness of the Transformed Parse Tree

A natural question regarding definitions such as those of the preceding section is whether the parse tree asserted to be the result of applying a transformation (or aggregate of transformations) is uniquely determined by the parse tree being transformed and the transformation. This section presents a proof that, by placing an order on the indefinite elements of environments (corresponding to top-down, left-to-right matching of indefinites in the SD of a transformation), the various predicates and consistency conditions act in concert to guarantee an affirmative answer to this question, except for generated symbols. (Indeed, the usefulness of generated symbols is precisely that successive generations of the same symbol may produce different results, as for the generated labels discussed in section 1.1.1.)

It should be clear that the definitions given so far do not always uniquely determine the match of a symbol in an indefinite list, in case the subtree being transformed contains multiple instances of a match for that symbol. That is, for Transformation 12 and example (14) of section 1.1.4, the definitions so far given do not determine that "j" will be the identifier first matched by <identifier>"1". For this particular example the order in which the matches occur is not significant, but it is possible to write transformations which depend on the order of matching indefinites, hence that order should be defined.

The first step is to define an order for two comparable indefinite

environment elements. Two environment elements are *comparable* if the predicate *is-comparable(e,e')* is satisfied:

$$\begin{aligned}
&\text{is-comparable}(e,e') = \text{is-G-env-elet}(e) \wedge \text{is-G-env-elet}(e') \\
&\quad \wedge \text{var}(e) = \text{var}(e') \wedge \text{s-parse}(e) = \text{s-parse}(e') \\
&\quad \wedge \neg\text{is-}\Omega\circ\text{s-indef-skel}(e) \wedge |\text{s-indef-skel}(e)| = |\text{s-indef-skel}(e')| \\
&\quad \wedge \underset{i=1}{\overset{|\text{s-indef-skel}(e)|}{\text{Et}}} (\exists\sigma_i)(\exists\sigma_i')[\sigma_i = \text{elem}(i)\circ\text{s-indef-skel}(e) \\
&\qquad \wedge\sigma_i' = \text{elem}(i)\circ\text{s-indef-skel}(e') \\
&\qquad \wedge\text{s-symbol}\circ\sigma_i\circ\text{s-parse}(e) = \text{s-symbol}\circ\sigma_i'\circ\text{s-parse}(e')]
\end{aligned}$$

That is, two environment elements are comparable if their variables and parse trees are equal, they both have indefinite skeletons, and corresponding elements of those skeletons select subtrees of the parse tree having the same head symbol.

If e_1 and e_2 are comparable environment elements, e_1 is less than e_2 if they satisfy the predicate *is-less-than* (e_1,e_2), where is-comparable (e_1,e_2):

$$\begin{aligned}
&\text{is-less-than}(e_1,e_2) = (\exists i)(\exists j_1)(\exists j_2)(\exists\sigma)(\exists\sigma_1)(\exists\sigma_2) \\
&\quad \left[\begin{array}{l} \overset{i-1}{(\text{Et}_{k=1}} \text{elem}(k)\circ\text{s-indef-skel}(e_1) = \text{elem}(k)\circ\text{s-indef-skel}(e_2)) \\ \wedge\text{elem}(i)\circ\text{s-indef-skel}(e_1) = \sigma_1\circ\text{elem}(j_1)\circ\sigma \\ \wedge\text{elem}(i)\circ\text{s-indef-skel}(e_2) = \sigma_2\circ\text{elem}(j_2)\circ\sigma \\ \wedge(j_1 < j_2 \vee [j_1 = j_2 \wedge \sigma_1 = \text{Id} \wedge \sigma_2 \neq \text{Id}])\end{array}\right]
\end{aligned}$$

That is, one of two comparable environment elements is less than the other if corresponding elements of their indefinite skeletons are equal up to some point, at which the selector of the first is either

left of, or properly contained in, the selector of the second.

Lemma 2.3.3 (1)

Let e_1 and e_2 be comparable environment elements. Then either $e_1 = e_2$ or one is less than the other.

Proof:

Suppose $e_1 \neq e_2$. Then by the definition of is-comparable e_1 and e_2 differ in their indefinite skeletons, i.e., $(\exists i')[\text{elem}(i') \circ \text{s-indef-skel}(e_1) \neq \text{elem}(i') \circ \text{s-indef-skel}(e_2)]$. Let i be the first such i', and for j=1,2, let $\tau_j = \text{elem}(i) \circ \text{s-indef-skel}(e_j)$. Now $\tau_1 \neq \tau_2$ implies either one is a proper head of the other, or they branch at some point. Suppose $\text{is-head}(\tau_1, \tau_2)$. Then $(\exists \sigma)(\exists \sigma_2)(\exists j)\ [\tau_1 = \text{elem}(j) \circ \sigma \wedge \tau_2 = \sigma_2 \circ \text{elem}(j) \circ \sigma \wedge \sigma_2 \neq \text{Id}]$ and hence $\text{is-less-than}(e_1, e_2)$. On the other hand, if τ_1 and τ_2 branch, $(\exists j_1)(\exists j_2)(\exists \sigma)(\exists \sigma_1)(\exists \sigma_2)\ [\tau_1 = \sigma_1 \circ \text{elem}(j_1) \circ \sigma \wedge \tau_2 = \sigma_2 \circ \text{elem}(j_2) \circ \sigma \wedge (j_1 < j_2 \vee j_2 < j_1)]$. But then $\text{is-less-than}(e_1, e_2) \vee \text{is-less-than}(e_2, e_1)$. Q.E.D.

Now suppose E is an environment and let $I_t(E)$ denote the subset of E consisting of those environment elements which correspond to indefinite patterns in the SD of a transformation t. The order for environment elements can be extended to an order on the subsets $I_t(E_1)$ and $I_t(E_2)$ of two environments E_1 and E_2, both of which suffice to match t to a parse tree p, by comparing the elements of these subsets in an order defined by t. The computation of the

relation between two environments is performed by the recursive predicate *is-not-greater-env*(E_1,E_2,*s-sd(t)*,*p*), where is-match(p,s-sd(t),E_1,{}) ∧ is-match(p,s-sd(t),E_2,{}):

```
is-not-greater-env(E1,E2,t,p)
   (is-FREE(t) ∨ is-final-pat(t) → T,
    is-prod-pat(t) → is-not-greater-env-list(E1, E2, s-sub-list(t),
       s-sub-list(p)),
    ( ∃e1)( ∃e2)[is-indef-pat(t) ∧ e1 ε E1 ∧ e2 ε E2 ∧ is-compat(e1,t)
       ∧ is-instance(p,e1,t) ∧ is-comparable(e1,e2) →
       (is-less-than(e1,e2) ∨ is-less-than(e2,e1) →
          ¬ is-less-than(e2,e1),
       T → is-not-greater-env-list(E1,E2,
                              |s-indef-list(t)|
            s-indef-list(t),        LIST   (elem(i)∘s-indef-skel(e1))(p))
                                    i=1
       )
   )

is-not-greater-env-list(E1,E2,t,p) =
   (|t| = 0 → T,
    is-not-greater-env(E1, E2, head(t), head(p))
    ∧ is-not-greater-env(E2, E1, head(t), head(p)) →
         is-not-greater-env-list(E1, E2, tail(t), tail(p)),
    T → is-not-greater-env(E1, E2, head(t), head(p))
   )
```

(Note that appropriate e_1 and e_2 always exist in is-not-greater-env, since both E_1 and E_2 match t and p; and that e_1 and e_2 are always comparable, since recursion terminates as soon as a pair are not equal.)

In analogy with lemma 2.3.3 (1) there is:

Lemma 2.3.3 (2)

Let E_1, E_2, t, and p be such that is-match(p, s-sd(t), E_1, {})$\wedge$ is-match(p, s-sd(t), E_2, {}). Then is-not-greater-env(E_1, E_2, s-sd(t), p) $\vee$ is-not-greater-env(E_2, E_1, s-sd(t), p), and both are true if and only if $I_t(E_1) = I_t(E_2)$.

The lemma follows immediately from the observation that, because s-sd(t) is finite, is-not-greater-env terminates. Also, is-not-greater-env(E_1,E_2,s-sd(t),p) $\wedge$ is-not-greater-env(E_2,E_1,s-sd(t), p) if and only if $\neg$is-less-than(e_1,e_2) $\wedge$ $\neg$is-less-than(e_2,e_1) for each corresponding pair of elements e_1 and e_2 in $I_t(E_1)$ and $I_t(E_2)$.

Since there are only a finite number of elements in $I_t(E)$ for any transformation t, and since there are only a finite number of distinct comparable environment elements with the same variable and parse tree, the following lemma holds:

Lemma 2.3.3 (3)

The collection $\{I_t(E_i)\}$ of I_t subsets of the elements of a collection $\{E_i\}$ of environments satisfying is-match(p,t,s-sd(t),E_i,{})is well ordered by is-not-greater-env.

There is thus an environment having a unique minimum I_t subset for each transformation-parse tree pair, as determined by the predicate is-indef-min (cf. 2.3.2 (1) Trlist Result and 2.3.2 (6) Subtrlist Result which use is-indef-min.)

Definition 2.3.3 (1) Indefinite Minimum

An environment E is *indefinite minimum* with respect to a parse tree p, transformation t, and global environment E' if the predicate *is-indef-min*(E,p,t,E') is satisfied, where is-G-env$(E \cup E')\wedge$ is-G-parse(p) $\wedge$ is-G-IGT(t):

$$\text{is-indef-min}(E,p,t,E') = \text{is-match}(p, \text{s-sd}(t), E' \cup E, \{\})$$
$$\wedge(\forall E'')[\text{is-G-env}(E' \cup E'') \wedge \text{is-match}(p, \text{s-sd}(t), E' \cup E'', \{\})$$
$$\supset \text{is-not-greater-env}(E, E'', \text{s-sd}(t), p)]$$

It is now possible to prove the uniqueness of the parse tree which is the result of applying a single transformation not containing subtransformations and applied in the context of an empty transformation set. For conciseness a transformation list of length one is constructed from the transformation and passed to is-trlist-result.

Theorem 2.3.3 (1)

Suppose is-G-IGT(t) $\wedge$ $(\forall\tau)$[is-pat$\circ\tau$(t) $\supset$ $|$s-subtrseq$\circ\tau$(t)$|$ = 0, is-G-parse(p), is-G-parse(p_1), and is-trlist-result(p_1, p, μ_0(<elem(1): t>), $\{\}$). Then p_1 is unique except possibly for subtrees corresponding to generable symbols in s-sc(t).

Proof:

By the definitions of is-trset-result and is-subtrlist-result, these predicates reduce to the equality predicate on their first two arguments for S = $\{\}$ and empty subtransformation sequences. Therefore, they may be ignored in is-match in this proof.

Suppose $(\exists p_2)$[is-trlist-result(p_2, p, μ_0(<elem(1): t>), {})]. Then p_2 satisfies is-tr-result for some indefinite minimum environment E_2, as does p_1 for E_1. But by Lemma 2.3.3 (3), $I_t(E_1)$ is uniquely determined by p and t, hence $I_t(E_1) = I_t(E_2)$.

Let $V_t(E) = \{e \in E \mid (\exists\tau)$[is-indexed$\circ\tau\circ$s-sd(t) $\wedge$ var(e) = var$\circ\tau\circ$ s-sd(t)]$\}$ (This excludes the generable symbols, which are those indexed symbols appearing in the SC of t but not in the SD.) Then $V_t(E_1) = V_t(E_2)$. For $V_t(E_1) \cap I_t(E_1) = V_t(E_2) \cap I_t(E_2)$, and if $e_i \in V_t(E_i) - I_t(E_i)$, i=1,2 and var($e_1$) = var($e_2$), by the definition of V_t $(\exists\tau)$[var(e_1) = var$\circ\tau\circ$s-sd(t)]. By definition of is-match, $(\exists\sigma_i)$[s-parse(e_i) = σ_i(p)], i=1,2. Now σ_i depends possibly on elements of $I_t(E_i)$, but $I_t(E_1) = I_t(E_2)$, hence $\sigma_1 = \sigma_2$; therefore, $e_1 = e_2$. Thus $V_t(E_1) = V_t(E_2)$.

Now it follows that $p_1 = p_2$ except for subtrees arising from elements of E_1-$V_t(E_1)$ and E_2-$V_t(E_2)$ (the generable symbols). For the tree structure of p_1 and p_2 is determined in three ways: from final patterns, production patterns, and indefinite patterns of t. Certainly the parts matching production patterns of t are identical, as are those determined from indefinite patterns since $I_t(E_1) = I_t(E_2)$ and the definition of is-G-IGT guarantees a unique corresponding symbol in s-sd(t) for each indefinite symbol in s-sc(t). As for final patterns of t, either these are unindexed, in which case by the definition of is-G-sc their symbol is a terminal symbol of G, as is that of p_1 and p_2 (by is-match) and all are equal; or else the final pattern is indexed, hence if the corresponding environment

element is in $V_t(E_1) = V_t(E_2)$, that subtree of p_1 and p_2 are equal. Q.E.D.

A similar theorem and proof are possible for subtransformations, where the scope rules of is-G-IGT guarantee that for all non-generable symbols in the SC there exist corresponding elements in the total environment $E' \cup E$. The theorem can then be extended to the result of applying a transformation set sequence, under the hypothesis that the sequence terminates. In view of the result of section 2.4, however, termination cannot always be guaranteed.

Corollary 2.3.2 (1)

Suppose is-G-IGT(t) $\wedge$ s-sd(t) = s-sc(t), and is-G-parse(p). Then is-trlist-result(p, p, μ_0(<elem(1): t>), {}).

Proof:

The proof follows immediately from the fact that the SD and SC of t satisfy both is-G-sd and is-G-sc. Hence t contains no subtransformations and no generable symbols, and theorem 2.3.3 (1) applies. Q.E.D.

2.4 Generative Power of Intra-Grammatical Transformations

This section is devoted to a proof that IGTs have at least the generative power of a simple Turing machine (in the sense of Davis [13]). The theorem is proved by stating an algorithm which, given the definition of a Turing machine, constructs an IGT system which simulates the operation of the Turing machine, in the sense that given the same initial instantaneous description they both give the same resultant.

To facilitate the proof, some definitions for IGTs paralleling those of [13] for Turing machines are introduced. Suppose that Z is a simple Turing machine with an alphabet $A = \{S_0, S_1, \ldots, S_n\}$, a set of states $\Sigma = \{q_1, q_2, \ldots, q_m\}$ and a set of quadruples K. A *transformational instantaneous description (TID) of* Z is a G_Z-parse tree whose head symbol is <inst descr>, where G_Z is the cfpsg:

$$G_Z = (\text{is-}G_Z\text{-nonterm}, \text{is-}G_Z\text{-term}, \text{is-}G_Z\text{-prod}, \langle\text{inst descr}\rangle)$$

and the relevant predicates are defined (in terms of the concrete forms of the elements they characterize) by:

$$\text{is-}G_Z\hat{}\text{-nonterm} = \{\langle\text{inst descr}\rangle, \langle\text{left part}\rangle, \langle\text{right part}\rangle, \langle\text{state}\rangle, \langle\text{symbol}\rangle\}$$

$$\text{is-}G_Z\hat{}\text{-term} = A \cup \Sigma \cup \{\#_L, \#_R\}$$

and the elements of $\text{is-}G_Z\hat{}$-prod are:

<inst descr> ::= <left part> <state> <right part>
<left part> ::= $\#_L$ | <left part> <symbol>
<right part> ::= $\#_R$ | <symbol> <right part>
<state> ::= q_1 | q_2 |...| q_m
<symbol> ::= S_0 | S_1 |...| S_n

Corresponding to the set of quadruples K of Z is constructed the <inst descr>-transformation list L_Z, according to the following algorithm:

(a) Corresponding to a quadruple of the form:

$$q_i\ S_j\ S_k\ q_\ell$$

add to L_Z the transformation:

'SD'
<left part> "1" q_i S_j <right part> "1"
==>
<left part> "1" q_ℓ S_k <right part> "1"
'SC'

(b) Corresponding to a quadruple of the form:

$$q_i\ S_j\ R\ q_\ell$$

add to L_Z the two transformations *in order:*

'SD'
<left part> "1" q_i S_j $\#_R$
==>
<left part> "1" S_j q_ℓ S_0 $\#_R$
'SC'
'SD'
<left part> "1" q_i S_j <right part> "1"
==>
<left part> "1" S_j q_ℓ <right part> "1"
'SC'

(c) Corresponding to a quadruple of the form:

$$q_i\ S_j\ L\ q_\ell$$

add to L_Z the two transformations:

'SD'

$\#_L\ q_i\ S_j$ <right part> "1"

==>

$\#_L\ q_\ell\ S_0\ S_j$ <right part> "1"

'SC'

'SD'

<left part> "1" <symbol> "1" $q_i\ S_j$ <right part> "1"

==>

<left part> "1" q_ℓ <symbol> "1" S_j <right part> "1"

'SC'

(These are the only kinds of quadruples which occur in K.)

A TID of Z is a *transformational terminal description (TTD) of* Z if it is the result of applying the transformation set sequence consisting of L_Z to a TID of Z.

If a TID ρ_2 is the result of applying a single transformation of L (which does apply) to a TID ρ_1, one writes ρ_1 ==> ρ_2. If ρ_p is a TTD of Z and, for $1 \leq i \leq p-1$, $\exists \rho_i$ such that ρ_i is a TID of Z and ρ_i ==> ρ_{i+1}, one writes $\rho_p = TR_Z(\rho_1)$, that is, ρ_p is the *transformational resultant* by Z of ρ_1.

With these definitions, the theorem may be stated (cf. section 2.2.2 for the definition of R):

Theorem 2.4 (1)

Let Z be a simple Turing machine and α_1, α_p be instantaneous descriptions of Z. Let G_Z and L_Z be a cfpsg and transformation list

constructed from Z as described above, and ρ_1, ρ_p be TIDs of Z, and suppose $R[\rho_1] = \#_L \alpha_1 \#_R$ and $R[\rho_p] = \#_L \alpha_p \#_R$. Then $\alpha_p = Res_Z(\alpha_1)$ if and only if $\rho_p = TR_Z(\rho_1)$.

Proof:

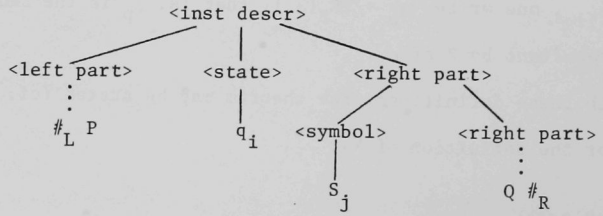
It suffices to prove that, for Z, $\alpha_n \rightarrow \alpha_{n+1}$ if and only if $\rho_n \Longrightarrow \rho_{n+1}$, where $R[\rho_n] = \#_L \alpha_n \#_R$ and $R[\rho_{n+1}] = \#_L \alpha_{n+1} \#_R$. By definition $\alpha_i \rightarrow \alpha_{i+1}$ implies one of five cases holds (cf. [13] p. 7); similarly by the algorithm above, $\rho_n \Longrightarrow \rho_{n+1}$ implies one of five similar cases (corresponding to the five kinds of transformations generated by the algorithm) holds.

Case (1). There exist expressions P and Q (possibly empty) such that:

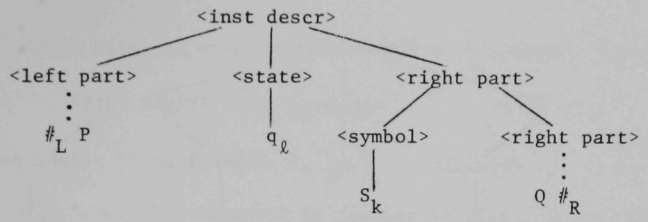
$$\alpha_n = P\ q_i\ S_j\ Q$$

$$\alpha_{n+1} = P\ q_\ell\ S_k\ Q$$

and $q_i\ S_j\ S_k\ q_\ell \in K$. Let ρ_n be a G_Z-parse tree such that $R[\rho_n] = \#_L \alpha_n \#_R$. Then by the definition of tape expression (definition 1.5 of [13]), it is clear that ρ_n corresponds to the tree:



Hence a transformation created in step (a) of the algorithm will apply to produce an object ρ_{n+1} corresponding to the tree:



so that $R[\rho_{n+1}] = \#_L\ P\ q_\ell\ S_k\ Q\ \#_R = \#_L\ \alpha_{n+1}\ \#_R$, as required. Moreover, there is only one transformation in L_Z which applies to ρ_n, since by the definition of Z there is only one quadruple in K beginning $q_i\ S_j \ldots$. Thus if $\rho_n \Longrightarrow \rho_{n+1}$, and $\#_L\alpha_n\#_R = R[\rho_n]$, then there is a quadruple:

$$q_i\ S_j\ S_k\ q_\ell$$

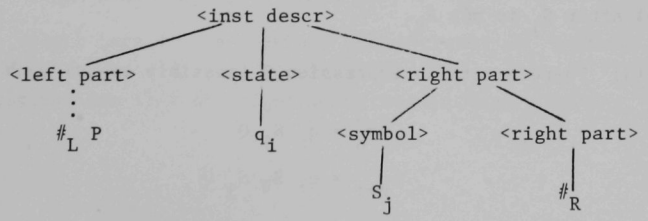
in K so that $\alpha_n \rightarrow \alpha_{n+1}$ where $\alpha_{n+1} = P\ q_\ell\ S_k\ Q$, or $\#_L\alpha_{n+1}\#_R = R[\rho_{n+1}]$.

(2) There exists an expression P (possibly empty) such that:

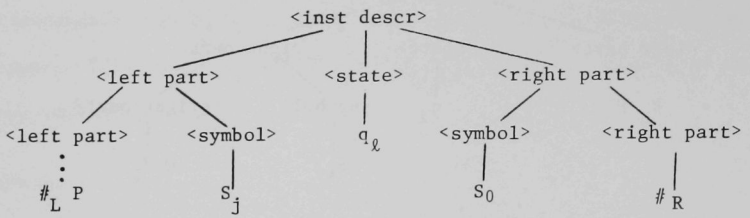
$$\alpha_n = P\ q_i\ S_j$$

$$\alpha_{n+1} = P\ S_j\ q_\ell\ S_0$$

and $q_i\ S_j\ R\ q_\ell \in K$. Let ρ_n be a G_Z-parse tree such that $R[\rho_n] = \#_L\alpha_n\#_R$. Then by definition, ρ_n corresponds to the tree:



Hence the *first* of a pair of transformations created in step (b) of the algorithm applies (note the importance of the order of this pair of transformations) to give an object ρ_{n+1} corresponding to the tree:



so that $R[\rho_{n+1}] = \#_L\ P\ S_j\ q_\ell\ S_0\ \#_R = \#_L\ \alpha_{n+1}\ \#_R$, as required. In this case there are two transformations in L_Z that apply to ρ_n, but the one ending in $\#_R$ is first in the list. Thus if $\rho_n \Longrightarrow \rho_{n+1}$ and $\#_L\alpha_n\#_R = R[\rho_n]$, it is this first transformation which will apply and its corresponding quadruple in K is $q_i\ S_j\ R\ q_\ell$, so that $\alpha_n \rightarrow \alpha_{n+1}$ where $\alpha_{n+1} = P\ S_j\ q_\ell\ S_0$, or $\#_L\ \alpha_{n+1}\ \#_R = R[\rho_{n+1}]$, as required.

(3) There exist expressions P and Q (possibly empty) such that:

$$\alpha_n = P\ q_i\ S_j\ S_k\ Q$$
$$\alpha_{n+1} = P\ S_j\ q_\ell\ S_k\ Q$$

and $q_i\ S_j\ R\ q_\ell \in K$. The argument is similar to case (2), but the second transformation of the pair of step (b) applies, since the symbol after S_j is not $\#_R$.

(4) There exists an expression Q (possibly empty) such that:

$$\alpha_n = q_i\ S_j\ Q$$
$$\alpha_{n+1} = q_\ell\ S_0\ S_j\ Q$$

and $q_i\ S_j\ L\ q_\ell \in K$. The argument is similar to case (1), and the first transformation of step (c) applies.

(5) There exist expressions P and Q (possibly empty) such that:

$$\alpha_n = P\ S_k\ q_i\ S_j\ Q$$

$$\alpha_{n+1} = P\ q_\ell\ S_k\ S_j\ Q$$

and $q_i\ S_j\ L\ q_\ell \in K$. The argument is similar to case (4), but the second transformation of the pair of step (c) applies.

Thus $\alpha_n \rightarrow \alpha_{n+1}$ if and only if $\rho_n \Longrightarrow \rho_{n+1}$, hence by induction, $\alpha_p = \mathrm{Res}_Z(\alpha_1)$ if and only if $\rho_p = \mathrm{TR}_Z(\rho_1)$ where $\#_L\alpha_1\#_R = R[\rho_1]$ and $\#_L\alpha_p\#_R = R[\rho_p]$. Q.E.D.

It is interesting to note that in this proof only the very simplest transformations (those with no indefinites and no subtransformations) are used. This is possible because the grammar G_Z is in some sense "well matched" to the task it must perform. One can gain some insight into the relationship between the form of the grammar and the need to use indefinites by considering the transformation list necessary to prove this theorem if the productions:

<right part> ::= $\#_R$ | <right part 1> $\#_R$

<right part 1> ::= <symbol> | <right part 1> <symbol>

are substituted for that of <right part> in the above grammar.

2.5 A Computer Implementation of Intra-Grammatical Transformations

The computer implementation of IGTs was constructed both to study the semantics of IGTs and to facilitate the design and verification of transformation set sequences. Both transformation set sequences and programs can be presented to the system in their concrete forms (as defined in section 2.2.2), and the transformed program is printed (or punched) by the system in its concrete form. This ability to work with the concrete forms of IGTs and programs (which are much simpler than their abstract forms) greatly facilitates use of the system.

The IGT system consists of two programs: the compiler, TRCOM, and the interpreter, XFORM. TRCOM is written in the Stanford compiler writing language XPL [28]. It inverts the mapping R of section 2.2.2 to translate the concrete forms of IGTs and programs into a LISP list representation of their abstract forms. The compiler is complete in the sense that it translates all of the features of transformations and subtransformations (imposing only the restriction that indices be integers in the range 0-999 and that mentioned at the end of section 2.5.2); however, it does not check all of the consistency conditions of section 2.2.1. (Thus certain syntactic errors, which are not detected in TRCOM, may be detected in XFORM as the transformations are interpreted.) TRCOM is parameterized in terms of the grammar used, hence a compiler for a different grammar may be obtained simply by replacing the grammar tables (produced by the XPL Analyzer program)

and the lexical analyzer, which processes input characters to convert them to terminal symbols of the grammar. TRCOM is not otherwise particularly innovative and will not be discussed further here.

XFORM is the transformation interpreter and is written in LISP [27]. It accepts a list representation of the pattern and parse trees which are the abstract forms of IGTs and programs and applies the former to the latter to produce the transformed program. In the following discussion the functions implementing XFORM are stated as LISP M-expressions employing the selectors and predicates introduced in sections 2.1, 2.2, 2.3, and the remainder of this section. (The actual LISP program used is written in terms of the explicit data representation chosen and includes checks for certain syntactic errors; it will not be discussed further here.)

The functions constituting XFORM may be divided into three categories: matching, changing, and sequencing. The matching functions determine if the SD of a given transformation matches a particular parse tree, and if it does, they produce the (indefinite minimum) environment defining the match. The changing functions then use this environment and the SC of the transformation to construct the transformed parse tree. Both matching and changing are performed under control of the sequencing functions.

Before the functions for matching, changing, and sequencing can be discussed, the representation for environments and certain synthesis functions for trees must be defined. Environments are, of course, represented by lists of environment elements. (Note that

a "dummy" NIL is added to the end of every environment to distinguish the empty environment from NIL, which indicates a transformation fails to match.) To look up entries in the environment, the following function is useful:

```
lookup[variable; env] ≡
    λ[[env elet];
        [null[env elet] → NIL;
         var[env elet] = variable → env elet;
         T → lookup[variable; cdr[env]]
        ]
     ] [car[env]]
```

Environment elements are list representations of the objects synthesized by the following function:

```
env-elet(v,p,s) =
    μ0(<s-symbol: s-symbol(v)>, <s-index: s-index(v)>,
       <s-parse: p>, <s-indef-skel: s>)
```

Here v is the result of the function var (cf. definition 2.2.1 (2)), p is a parse tree, and s is an indefinite skeleton represented by "severely pruning" a copy of the parse tree of the element so that it extends only to the substitutable nodes, i.e., the nodes selected by the selectors in the indefinite skeleton. This representation of an indefinite skeleton, which for the remainder of this section is considered to be selected by s-indef-skel, may be taken to be the list representation of the abstract object defined by the predicate *is-G-skel*:

is-skel = is-final-skel $\vee$ is-prod-skel

is-final-skel = (<s-symbol: is-symbol>, <s-subst: T $\vee$ F>)

is-prod-skel = (<s-symbol: is-symbol>, <s-sub-list: is-skel-list>)

is-G-skel(s) = is-skel(s)

$\wedge(\forall\tau)$[is-final-skel$\circ\tau$(s) $\supset$ is-G-symbol$\circ$s-symbol$\circ\tau$(s)

$\wedge$is-prod-skel$\circ\tau$(s) $\supset$ is-G-prod$\circ$immed-prod$\circ\tau$(s)

$\wedge(\exists\tau')$[s-subst$\circ\tau'\circ\tau$(s)]]

(Note that the last conjunct defines the condition terminating "pruning".)

Finally, the group of functions which synthesize parse trees and skeleton trees are introduced. These are:

tree(s,ℓ) = μ_0(<s-symbol: s>, <s-sub-list: ℓ>)

where is-G-symbol(s) $\wedge$ (is-parse-list(ℓ) $\vee$ is-skel-list(ℓ));

final-skel(s,f) = μ_0(<s-symbol: s>, <s-subst: f>)

where is-G-symbol(s) $\wedge$ (f $\vee\neg$f);

final-parse(s) = μ_0(<s-symbol: s>)

where is-G-term(s). The LISP functions *cons*, *car*, and *cdr* are used for operations on lists.

2.5.1 Functions for Matching

The functions which perform the matching operation are *match*, *matchlist*, *matchindefinite*, *matchindefinitelist*, *finish*, and *checkmatch*, assisted by *join*, *tree1*, and *lookup*.

The key to understanding the matching operation is the concept of *failure* of a pattern tree to match a parse tree. Failure for final and production patterns is quite simple: the former fails if its symbol and that of the parse tree are not equal; the latter fails for the same reason, or if its sublist and that of the parse tree are not the same length, or if some pattern tree in the sublist fails. They also fail if they are indexed, a match of their variable has previously occurred, and the parse tree is not equal to that previously matched.

Failure for indefinites is more complex, since the pattern trees in an indefinite list undergo conditional matching. Thus a pattern tree in an indefinite list may fail for the reasons given above or because an adjacency relation (cf. section 2.3.1) is violated; if not, it conditionally matches. This conditional match, however, may ultimately fail because a pattern tree later in the indefinite list fails, or because some pattern tree later in the SD fails. If a conditional match fails, the match is reattempted later in the parse tree, subject to the satisfaction of the adjacency relations for indefinites. (Here "later" is to be understood as "further down, or further to the right".)

The matching functions are defined below. The identifiers used are largely self-explanatory, except for *rest of sd*, *rest of parse*, and *unmatched*. The concept of failure outlined above necessitates main-

taining the matches of all pattern trees in indefinite lists as an unbroken sequence of recursive function calls until the entire SD has been matched. Therefore, auxiliary stacks are needed to pass down unexamined parts of the SD and parse tree; these are *rest of sd* and *rest of parse*. *Unmatched* implements the consistency condition for indefinite trees by indicating when no substitutable skeleton has yet been added to a skeleton list. All of the functions return a list of the form *cons*[*X*,*env*], where *X* is the atom T if no indefinite pattern has been matched lower in the pattern tree. If such an indefinite pattern has been matched, X is a list of one or more partially constructed skeleton trees.

```
match[sd; parse; env; rest of sd; rest of parse] ≡
    [s-symbol[parse] = s-symbol[sd] →
          checkmatch[var[sd]; ¬is-indef-pat[sd], parse;
            [is-final-pat[sd] → env;
             is-indef-pat[sd] → matchindefinitelist[s-indef-list[sd];
                    list[parse]; env; F; rest of sd; cons[DNI;
                    rest of parse]];
             T → matchlist[s-sub-list[sd]; s-sub-list[parse]; env;
                    rest of sd; rest of parse]
            ]];
     T → NIL
    ]


matchlist[sd list; parse list; env; rest of sd; rest of parse] ≡
    [null[sd list] → [null[parse] → env; T → NIL];
     null[parse] → NIL;
     T → λ[[envl];
              [null[envl] → NIL;
               car[envl] → matchlist[cdr[sd list]; cdr[parse list];
                      envl; rest of sd; rest of parse];
               T → envl
              ]
         ] [match[car[sd list]; car[parse list]; env;
              cons[cdr[sd list]; rest of sd]; cons[cdr[parse list];
              rest of parse]]]
    ]
```

```
matchindefinite[indef list; parse; env; rest of sd; rest of parse] ≡
    λ[[indef list1];
        [null[indef list1] → NIL;
         T → λ[[env1];
                 [null[env1] → treel[s-symbol[parse];
                         matchindefinitelist[indef list;
                         s-sub-list[parse]; env; T; rest of sd;
                         rest of parse]];
                  car[env1] →
                      λ[[env2];
                           [null[env2] → treel[s-symbol[parse];
                                   matchindefinitelist[indef list;
                                   s-sub-list[parse]; env; T;
                                   rest of sd; rest of parse]];
                             T → join[final-skel[s-symbol[parse];
                                   T]; env2]
                           ]
                       ] [matchindefinitelist[cdr[indef list1];
                            car[rest of parse]; env1; F;
                            rest of sd; cdr[rest of parse]]];
                  T → join[final-skel[s-symbol[parse]; T]; env1]
                 ]
             ] [match[car[indef list1]; parse; env,
                  cons[cons[IND; cdr[indef list1]]; rest of sd];
                  rest of parse]]
        ]
    ] [ [is-FREE[car[indef list]] → cdr[indef list];
         T → indef list
        ]]
```

```
matchindefinitelist[indef list; parse list; env; unmatched;
        rest of sd; rest of tree] ≡
    [null[parse list] →
        [unmatched → NIL;
         T → join[NIL; matchindefinitelist[indef list;
                car[rest of parse]; env; F; rest of sd;
                cdr[rest of parse]]];
        ];
     is-DNI[parse list] →
        [null[indef list] ∨ is-FREE[car[indef list]]
           ∧null[cdr[indef list]] →
             finish[car[rest of sd]; car[rest of parse]; env;
               cdr[rest of sd]; cdr[rest of parse]];
         T → NIL
        ];
    T → λ[[envl];
          [is-FREE[car[indef list]] ∧ null[envl] →
               join[final-skel[s-symbol[car[parse list]]; F];
                 matchindefinitelist[indef list; cdr[parse list];
                 env; unmatched; rest of sd; rest of parse]];
           T → envl
          ]
        ] [matchindefinite[indef list; car[parse list]; env;
             rest of sd; cons[cdr[parse list]; rest of parse]]]
  ]
```

```
finish[sd list; parse list; env; rest of sd; rest of parse] ≡
    [null[rest of sd] →
        [null[rest of parse] → cons[NIL; cdr[env]];
         T → NIL
        ];
     null[rest of parse] → NIL;
     null[sd list] →
        [null[parse list] → finish[car[rest of sd];
                car[rest of parse]; env; cdr[rest of sd];
                cdr[rest of parse]];
         T → NIL
        ];
     T → λ[[envl];
           [null[envl] → NIL;
            car[envl] → finish[car[rest of sd];
                   car[rest of parse]; envl; cdr[rest of sd];
                   cdr[rest of parse]];
            T → envl
           ]
        ] [ [is-IND[car[sd list]] →
                   matchindefinitelist[cdr[sd list]; parse list;
                   env; F; rest of sd; rest of parse];
            T → matchlist[sd list; parse list; env; rest of sd;
                   rest of parse]
           ]]
    ]
```

```
checkmatch[variable; nonindef; parse; env] ≡
    [null[env] → NIL;
     is-Ω[s-index[variable]] →
         [nonindef → env;
          T → cons[cdr[car[env]]; cdr[env]]
         ];
     T → λ[[env elet];
              [null[env elet] → cons[
                       [nonindef → car[env]; T → cdr[car[env]]];
                       cons[env-elet[variable; parse;
                       [nonindef → NIL; T → car[car[car[env]]]]];
                       cdr[env]]];
               parse = s-parse[env elet] → env;
               T → NIL
              ]
          ] [lookup[variable; cdr[env]]]
    ]

join[skel; env] ≡
    [null[env] → NIL;
     null[skel] → cons[cons[skel; car[env]]; cdr[env]];
     T → cons[cons[cons[skel; car[car[env]]]; cdr[car[env]]];
           cdr[env]]
    ]

treel[symbol; env] ≡
    [null[env] → NIL;
     T → cons[cons[cons[tree[symbol; car[car[env]]];
           car[cdr[car[env]]]]; cdr[cdr[car[env]]]]; cdr[env]]
    ]
```

2.5.2 Functions for Changing

The functions which perform the changing operation are *change*, *changelist*, *changeindefinite*, and *changeindefinitelist*, assisted by *lookup*, *applytrset*, and *applysubtrseq*. These latter two functions are sequencing functions and are described in section 2.5.3. For the purpose of understanding the changing functions, they may be considered to be the identity function on their first argument.

The operation of the changing functions is straightforward. Note that the functions for indefinites employ the indefinite skeleton of an environment element as a kind of "template" to locate the nodes in the parse tree of that element where the parse trees corresponding to non-free symbols in the indefinite list must be substituted.

The changing functions are defined below; again the identifiers used are self-explanatory. The functions in all cases return the partially constructed parse tree, which in the case of the indefinite functions is prefixed by the unused remainder of the indefinite list of the indefinite pattern guiding construction.

```
change[sc; env; tr set] ≡
    [is-indexed[sc] →
        λ[[env elet];
            [null[env elet] → s-parse[car[cdr[rplacd[env; cons
                    [env-elet[var[sc]; generate[s-symbol[sc]];
                    NIL]; cdr[env]]]]]];
             is-indef-pat[sc] →
                    cdr[changeindefinite[s-indef-list[sc];
                    s-indef-skel[env elet]; s-parse[env elet];
                    env; tr set]];
             T → s-parse[env elet]
            ]
         ] [lookup[var[sc]; env]];
     is-final-pat[sc] → sc;
     T → tree[s-symbol[sc]; changelist[s-sub-list[sc]; env; tr set]]
    ]


changelist[sc list; env; tr set] ≡
    [null[sc list] → NIL;
     T → cons[applytrset[applysubtrseq[change[car[sc list]; env;
            tr set]; s-subtrseq[car[sc list]]; env; tr set];
            tr set]; changelist[cdr[sc list]; env; tr set]]
    ]
```

```
changeindefinite[indef list; skel; parse; env; tr set] ≡
    λ[[indef list1];
        [is-final-skel[skel] →
            [s-subst[skel] → cons[cdr[indef list1];
                    applysubtrseq[change[car[indef list1]; env;
                    tr set]; s-subtrseq[car[indef list1]]; env;
                    tr set]]:
             T → cons[indef list1; parse]
            ]
         T → λ[[indef cum parse list];
                  cons[car[indef cum parse list];
                  tree[s-symbol[skel]; cdr[indef cum parse list]]]
              ] [changeindefinitelist[indef list1;
                  s-sub-list[skel]; s-sub-list[parse]; env;
                  tr set]]
        ]
     ] [ [null[indef list] ∨ ¬is-FREE[car[indef list]]] → indef list;
         T → cdr[indef list]
       ]]


changeindefinitelist[indef list; skel list; parse list; env; tr set] ≡
    [null[skel list] → cons[indef list; NIL];
     T → λ[[indef cum parse];
            λ[[indef cum parse list];
                  cons[car[indef cum parse list];
                  cons[applytrset[cdr[indef cum parse]; tr set];
                  cdr[indef cum parse list]]]
             ] [changeindefinitelist[car[indef cum parse];
                  cdr[skel list]; cdr[parse list]; env; tr set]]
         ] [changeindefinite[indef list; car[skel list];
              car[parse list]; env; tr set]]
    ]
```

One should note that the manner in which generated symbols are implemented in *change* is not quite faithful to the semantics of generable symbols as given by the definitions of sections 2.2 and 2.3. This may be seen by considering a generable symbol which occurs in the SC of a transformation, and also in the SC of a subtransformation in it. If these symbols have the same index, they should represent the same parse tree, but this will not be the case for the above definition of *change* if the symbol occurs later in the SC of the transformation than does the subtransformation which uses it. This discrepancy could easily be corrected by prefacing the sc-tree of each transformation (and subtransformation) by a list of the generable symbols appearing in it, and generating the corresponding parse trees for these symbols and adding them to the environment before starting the changing operation. For the examples of section 3 this discrepancy has no effect, since in no case does the above condition hold.

2.5.3 Functions for Sequencing

The functions which perform the sequencing operation are: *applysubtrlist*, *searchlist*, *applytrset*, *searchset*, *applytrsetrec*, *applytrsetreclist*, *applytrsetseq*, and *applysubtrseq*.

Those functions whose names begin with "apply" correspond roughly to the predicates of section 2.3.2 whose names end in "-result". However, the function *applysubtrlist* corresponds to both of the predicates is-subtrlist-result and is-trlist-result. It has the basic form of is-subtrlist-result (with global environment and Markov sequencing). When it is employed for is-trlist-result, it is supplied the null environment, and advantage is taken of the equivalence of Markov sequencing and recursive reapplication of the transformation set for the top node of the parse tree (cf. the discussion in section 1.1.2). The functions *searchlist* and *searchset* locate the appropriate element of a transformation (or subtransformation) list and transformation set, respectively. (Transformation sets, of course, are represented by a list of transformation lists.)

The application of a transformation set sequence to a parse tree is initiated by calling *applytrsetseq*.

```
applysubtrlist[parse; tr subtr list; global env; tr set] ≡
    λ[[changed parse];
        [null[changed parse] → parse;
         T → applysubtrlist[changed parse; tr subtr list;
                global env; tr set]
        ]
     ] [searchlist[parse; tr subtr list; global env; tr set]]
```

```
searchlist[parse; tr subtr list; global env; tr set] ≡
    [null[tr subtr list] → NIL;
     T → λ[[env];
            [null[env] → searchlist[parse; cdr[tr subtr list];
                   global env; tr set];
             T → applysubtrseq[change[s-sc[car[tr subtr list]];
                   env; tr set]; s-subtrseq[s-sc[car[
                   tr subtr list]]]; env; tr set]
            ]
          ] [match[s-sd[car[tr subtr list]]; parse; global env;
              (NIL); (NIL)]]
    ]

applytrset[parse; tr set] ≡
    λ[[tr list];
        [null[tr list] → parse;
         T → applysubtrlist[parse; tr list; (NIL); tr set]
        ]
     ] [searchset[s-symbol[parse]; tr set]]

searchset[symbol; tr set] ≡
    [null[tr set] → NIL;
     s-symbol[s-sd[car[car[tr set]]]] = symbol → car[tr set];
     T → searchset[symbol; cdr[tr set]]
    ]
```

```
applytrsetrec[parse; tr set] ≡
    λ[[tr list];
        [null[tr list] → tree[s-symbol[parse];
                applytrsetreclist[s-sub-list[parse]; tr set]];
         T → applysubtrlist[tree[s-symbol[parse];
                applytrsetreclist[s-sub-list[parse]; tr set]];
                tr list; (NIL); tr set]
        ]
     ] [searchset[s-symbol[parse]; tr set]]

applytrsetreclist[parse list; tr set] ≡
    [null[parse list] → NIL;
     T → cons[applytrsetrec[car[parse list]; tr set];
            applytrsetreclist[cdr[parse list]; tr set]]
    ]

applytrsetseq[parse; tr set seq] ≡
    [null[tr set seq] → parse;
     T → applytrsetseq[applytrsetrec[parse; car[tr set seq]];
            cdr[tr set seq]]
    ]

applysubtrseq[parse; subtr seq; global env; tr set] ≡
    [null[subtr seq] → parse;
     T → applysubtrseq[applysubtrlist[parse; car[subtr seq];
            global env; tr set]; cdr[subtr seq]; global env; tr set]
    ]
```

3. Applications of Intra-Grammatical Transformations

In this section I discuss the application of intra-grammatical transformations to the solution of two programming language definitional problems: the problem of (static) identifier denotation, which is the problem of associating an appearance of a variable in a program with the applicable declaration of that variable; and the problem of for statement optimization, which is the problem of determining when certain computations can be moved from a point within the controlled statement of a for statement to a point outside it, and so moving them.

In both cases the solutions are given for a slightly modified version of the language Algol 60, appropriately extended to cover new constructions introduced in the solution of these problems. The grammar for this language is listed in section 6.1. In addition to the extensions necessary for these examples (discussed in sections 3.1 and 3.2), modifications were made to the Algol 60 grammar as given in the Revised Report [30] to make it compatible with the Stanford XPL compiler generator system [28] used to construct a parser for programs and transformations. These modifications are summarized at the beginning of section 6.1. Throughout the remainder of this section the use of the grammar of section 6.1 is to be understood. (However, symbols such as 'BEGIN', enclosed in escape symbols in section 6.1, will continue to be written with underlining (begin), following the usual Algol 60 convention.)

3.1 Transformations for Identifier Denotation

The problem of identifier denotation in Algol 60 is the problem of associating each occurrence of an identifier in a program with the appropriate declaration of that identifier, taking into account the scope rules of declarations. This problem may be divided into static and dynamic identifier denotation, the former depending on the lexicographic order of occurrence of blocks in the program, and the latter depending on the order of execution of the blocks. It is the static identifier denotation problem which is treated here. (In view of the theorem of section 2.4, a transformational specification of dynamic identifier denotation is also possible, but it is likely to be considerably less elegant than that of static denotation since the denotation of recursive procedure calls requires that it either interact with the specification of program semantics or specify them itself. For an idea of the complication involved, see de Bakker [4].) The following subsection describes the theory underlying the identifier denotation transformations, and section 3.1.2 describes the transformations themselves.

3.1.1 Theory of Identifier Denotation

The solution for the static identifier denotation problem developed here is essentially a transformational formulation of the algorithms for static identifier denotation described by Boyle and Grau [6] (except that no attempt is made to maintain the ordering of serial numbers as is done in those algorithms). Since reference [6] contains an extensive discussion of the identifier denotation problem and a review of the pertinent literature, these will not be repeated here.

Identifier denotation is carried out by substituting for each <identifier token> in a block a <denotation> (cf. section 6.1) consisting of a <serial number> (as always, distinct from all previously used serial numbers) and a <type indicator> which is derived from the declaration of the identifier. The declaration itself is replaced by dec followed by the appropriate <denotation>. In the case of array, switch, and procedure declarations, the information which they contain in addition to type is moved to the <compound tail> of the block. Thus after application of the identifier denotation transformations to a program, each <block head> in the program consists simply of begin followed by a list of declarations (separated by semicolons) of the form dec <denotation>.

For the denotation of a <simple variable> an appropriately simple transformation suffices, assuming that all <type declarations>s have been transformed so that their <type list>s contain only one element:

```
<block>{
'SD'
   <block head> "1"
   { ? <declaration> { <type> "1" <identifier> "1" } ? } ;
   <compound tail> "1"
==>
   <block>
   { <block head> "1"
      { ? <declaration> { dec <serial number> "1"
           <type> "1" } ? } ;
      <compound tail> "1"
   }{
   'SD'
      <block> "11" { ? <identifier> "1" ? }
   ==>
      <block> "11"
      { ? <identifier> { <serial number> "1" <type> "1" } ? }
   'SC'}
'SC'}
```

The denotation of arrays is somewhat more complicated than that of simple variables, since the information in the <bound pair list> of the array declaration must be removed from the declaration. This information is placed in the <actual parameter list> of an explicit call to the (new) standard function *allocate*, executed for its side effect, which is to allocate the required amount of storage indicated by its parameters. (It is assumed that *allocate* is capable of accepting a variable number of parameters.) Thus the following block containing an array declaration:

```
begin array A[1:n, 1:m, -2×k:2×k];
      A[1,1,0] := 1                                    (1)
end
```

is replaced by:

```
begin dec #1 real array;
      allocate (#1 real array, 1, n, 1, m, -2×k, 2×k);
      #1 real array [1, 1, 0] := 1
end
```

(The transformation lists involved in this conversion are the <array segment>-, <array list>-, <block head>-, and <block>-transformation lists.)

The denotation of labels is quite similar to that of simple variables, but requires the creation of a declaration for the serial number denoting the label, since, of course, there are no label declarations in Algol 60. Another aspect of label denotation is concerned with implementing the restriction of section 4.6.6 of the Revised Report, which requires that labels labeling statements within the controlled statement of a for statement be local to the controlled statement. This restriction is implemented by searching the controlled statement for a (undenoted) label definition, and if one is found, converting the controlled statement into a block.

Switches are denoted in a manner similar to that of arrays, with the expressions constituting the <switch list> being moved into the <actual parameter list> of *allocate*. (Here *allocate* is assumed to reserve the necessary storage locations and initialize them to

the corresponding expressions.)

The denotation of procedure declarations is the most complicated aspect of identifier denotation. As discussed in [6], it begins by imbedding the <procedure body> in a block containing a result declaration for the procedure identifier; this insures that the <procedure body> behaves as if it were a block, as required by section 5.4.3 of the Revised Report, and also provides an identifier to receive the result of a function procedure.

For a formal parameter called by value, a corresponding local variable is declared with a <denotation> whose <type indicator> is derived from the <specification part> of the procedure, and this <denotation> is substituted for the identifier of the formal parameter throughout the <procedure body>. If the formal parameter is an array called by value, this <denotation>, followed by the formal parameter itself, is placed in the <actual parameter list> of a call to *allocate*. (Here *allocate* is assumed to reserve an amount of storage corresponding to that of the formal parameter and to initialize the elements of it to the values of the corresponding elements of the formal parameter.) If the formal parameter called by value is not an array, the <denotation> is placed in the <left part> of an assignment statement whose <expression> is the formal parameter itself.

Finally, each formal parameter (whether called by name or by value) is declared with a <denotation> of the form <serial number> formal, and is so denoted throughout the <procedure body>. (Of course, for parameters called by value this <denotation> only appears

in the declaration and the statement assigning the value to the local identifier described above.)

Thus the following procedure:

```
integer procedure p(a,b,c);
value a,b;  real a;  array b;                                    (2)
p := if c ≤ 0 then 1 else entier (a+b[1]×c) + p(a,b,c-1);
```

is transformed into:

```
integer proc p;
begin dec #1 integer result;
      dec #2 real array;
      dec #3 real;
      dec #4 formal;
      dec #5 formal;
      dec #6 formal;
      #3 real := #4 formal;
      allocate (#2 real array, #5 formal);
      #1 integer result := if #6 formal ≤ 0 then 1 else
          entier (#3 real + #2 real array [1] × #6 formal)
          + p (#3 real, #2 real array, #6 formal -1);
      return #1 integer result
end
```

(The above changes are accomplished by the <procedure declaration>-transformation list.) The final step in denotation of a procedure declaration (performed by a member of the <block>-transformation list) is to denote the procedure identifier throughout the block, and move the procedure body to a skip statement at the beginning of the <compound tail> of the block. (The skip statement is assumed

to cause skipping of the execution of the <procedure body> upon entry to the block.)

The discussion of the theory of identifier denotation is thus complete, except to note that the bottom-up sequencing rule for transformation sets guarantees that the identifiers in nested blocks will be denoted in the proper order, thus automatically implementing the identifier scope rules of Algol 60 (cf. Section 6.2).

3.1.2 The Identifier Denotation Transformation Set

At the end of this section is listed the identifier denotation transformation set used with the computer implementation of IGTs to perform identifier denotation. The transformation lists comprising this transformation set and their contribution to identifier denotation will be discussed briefly.

The <block head>-transformation list converts type and array declarations to "single form", in which each contains the declaration of only one identifier. Also, it replaces all occurrences of the abbreviation array in declarations by its full form real array.

The <array segment>-transformation list converts the <bound pair list> of an array declaration into a call on the *allocate* standard function. This is done "in place" by placing the call to *allocate* in a dummy <bound pair>.

The <array list>-transformation list helps the conversion of array declarations to single form by replicating the bounds information among all identifiers in an <array segment>. Note that the syntax insures that the <array segment>-transformation list applies before the replication of the bounds information takes place, thus minimizing the amount of transformation performed.

The operation of the <procedure declaration>-transformation list is discussed in section 3.1.1. Note that it contains separate transformations for functions and "proper" procedures; this minimizes

the use of indefinites.* The subtransformations:

<specifier>{'SD' array ==> real array 'SC'}

in subtransformations 1.B.1 and 2.A.1 are an alternative to creating a <specifier>-transformation list to make this change. (The latter would be highly inefficient because of the number of <specifier>s created as <type indicator>s during identifier denotation.)

The <block>-transformation list performs the actual denotation of the various kinds of identifiers (cf. section 3.1.1).

The <for statement>-transformation list insures that labels defined in the controlled statement of a for statement are local to it (cf. section 3.1.1). After the block is created and the first such label is denoted, the remainder are denoted by the <block>-transformation list.

The following points should be noted about this transformation set: First, it does not attempt to "diagnose" violations of the Algol 60 scope rules. This lack is not serious except for local identifiers occurring in the <bound pair list> of an array declaration, which will be denoted in the same manner as if they had occurred in a procedure or switch declaration. This scope violation could be trapped by adding at the end of the <block>-transformation list a transformation which converts any <denotation>s in the second or following parameters of *allocate* to a standard identifier indicating this error had occurred (taking care not to destroy switch and value

*Subtransformations 1.B.1 and 2.A.1 are already the most complex transformations (in terms of indefinites) in the examples of section 3.

array calls to *allocate*, however). The <identifier token>s of undeclared identifiers, including those of identifiers used outside the scope of their declarations, of course remain in the program after all properly declared <identifier token>s have been replaced by <denotation>s; they are thus self diagnosing (cf. section 6.2).

Second, this transformation set does not denote own variables. This is a consequence of a "glitch" in the grammar of section 6.1 and could be corrected by adding a production:

<type indicator> ::= own <type> | own <type> array

and making appropriate modifications to the various transformation lists. As the static denotation of own variables need not differ from that of non-own variables, this change was not deemed worthwhile.

The transformation lists comprising the identifier denotation transformation set follow below; examples of its application to programs are given in section 6.2.

```
'COMMENT' BLKHDTRS VERSION1.  TRANSFORMATIONS TO CONVERT DECLARATIONS
    TO SINGLE FORM;

<BLOCK HEAD>
@

'COMMENT' 1.  CONVERT TYPE DECLARATIONS TO SINGLE FORM;

'SD'
    <BLOCK HEAD> "1" ; <LOCAL OR OWN TYPE> "1" <SIMPLE VARIABLE> "1" ,
    <TYPE LIST> "1"
==>
    <BLOCK HEAD> "1" ; <LOCAL OR OWN TYPE> "1" <SIMPLE VARIABLE> "1" ;
    <LOCAL OR OWN TYPE> "1" <TYPE LIST> "1"
'SC'

'COMMENT' 2.  CONVERT ALL ARRAY DECLARATIONS TO SINGLE FORM;

'SD'
    <BLOCK HEAD> "1" ;
    <LOCAL OR OWN TYPE> "1" 'ARRAY' <ARRAY LIST> "1" ,
        <ARRAY SEGMENT> "1"
==>
    <BLOCK HEAD> "1" ;
    <LOCAL OR OWN TYPE> "1" 'ARRAY' <ARRAY LIST> "1" ;
    <LOCAL OR OWN TYPE> "1" 'ARRAY' <ARRAY SEGMENT> "1"
'SC'

'COMMENT' 3.  CONVERT TYPE DECLARATIONS TO SINGLE FORM;

'SD'
    'BEGIN' <LOCAL OR OWN TYPE> "1" <SIMPLE VARIABLE> "1" ,
    <TYPE LIST> "1"
==>
    'BEGIN' <LOCAL OR OWN TYPE> "1" <SIMPLE VARIABLE> "1" ;
    <LOCAL OR OWN TYPE> "1" <TYPE LIST> "1"
'SC'

'COMMENT' 4.  CONVERT ALL ARRAY DECLARATIONS TO SINGLE FORM;

'SD'
    'BEGIN'
    <LOCAL OR OWN TYPE> "1" 'ARRAY' <ARRAY LIST> "1" ,
        <ARRAY SEGMENT> "1"
==>
    'BEGIN'
    <LOCAL OR OWN TYPE> "1" 'ARRAY' <ARRAY LIST> "1" ;
    <LOCAL OR OWN TYPE> "1" 'ARRAY' <ARRAY SEGMENT> "1"
'SC'
```

```
'COMMENT' 5.  CONVERT ARRAY TO REAL ARRAY;

'SD'
    <BLOCK HEAD> "1" ; 'ARRAY' <ARRAY LIST> "1"
==>
    <BLOCK HEAD> "1" ; 'REAL' 'ARRAY' <ARRAY LIST> "1"
'SC'

'COMMENT' 6.  CONVERT ARRAY TO REAL ARRAY;

'SD'
    'BEGIN' 'ARRAY' <ARRAY LIST> "1"
==>
    'BEGIN' 'REAL' 'ARRAY' <ARRAY LIST> "1"
'SC'
%
```

```
'COMMENT' ARSEGTRS VERSION3.  TRANSFORMATIONS TO CREATE ALLOCATE CALLS
    FROM THE BOUND PAIRS IN AN ARRAY SEGMENT;

<ARRAY SEGMENT>
@

'COMMENT' 1, PLACE MARKER TO TERMINATE APPLICATION OF <ARRAY SEGMENT>
    TRANSFORMATIONS;

'SD'
    <IDENTIFIER> "1" (/ <BOUND PAIR LIST> "1"
    @ 1 : ALLOCATE ( <ACTUAL PARAMETER LIST> ) % /)
==>
    <IDENTIFIER> "1" '(/1' <BOUND PAIR LIST> "1" /)
'SC'

'COMMENT' 2, MOVE UPPER AND LOWER BOUNDS TO THE FORMAL PARAMETER
    LIST OF THE ALLOCATE PROCEDURE;

'SD'
    <IDENTIFIER> "1" (/ <BOUND PAIR LIST> "1"
    @ <BOUND PAIR LIST>
        @ 1 : ALLOCATE ( <ACTUAL PARAMETER LIST> "1" ) ,
            <EXPRESSION> "1" : <EXPRESSION> "2"
        % ?
    % /)
==>
    <IDENTIFIER> "1" (/ <BOUND PAIR LIST> "1"
    @ <BOUND PAIR LIST>
        @ 1 : ALLOCATE ( <ACTUAL PARAMETER LIST> "1" ,
            <EXPRESSION> "1" , <EXPRESSION> "2" )
        % ?
    % /)
'SC'

'COMMENT' 3, CREATE AN ALLOCATE CALL IN A BOUND PAIR LIST;

'SD'
    <IDENTIFIER> "1" (/ <BOUND PAIR LIST> "1"
    @ <BOUND PAIR HEAD> @ <EXPRESSION> "1" : % <EXPRESSION> "2" ? % /)
==>
    <IDENTIFIER> "1" (/ <BOUND PAIR LIST> "1"
    @ <BOUND PAIR HEAD> @ <EXPRESSION> @ 1 % : % <EXPRESSION>
        @ ALLOCATE ( <IDENTIFIER> "1" , <EXPRESSION> "1" ,
            <EXPRESSION> "2" )
        % ?
    % /)
'SC'
%
```

```
'COMMENT' ARLSTTRS VERSION1.  TRANSFORMATIONS TO ASSOCIATE A BOUND PAIR
    LIST WITH EACH IDENTIFIER IN AN ARRAY SEGMENT;

<ARRAY LIST>
@

'COMMENT' 1.  CONVERT THE ARRAY SEGMENT TO SINGLE FORM;

'SD'
    <ARRAY LIST> "1" , <IDENTIFIER> "1" , <ARRAY SEGMENT> "1"
    @ ? '(/1' <BOUND PAIR LIST> "1" /) %
==>
    <ARRAY LIST> "1" , <IDENTIFIER> "1" '(/1' <BOUND PAIR LIST> "1" /) ,
    <ARRAY SEGMENT> "1"
'SC'

'COMMENT' 2.  CONVERT THE ARRAY SEGMENT TO SINGLE FORM;

'SD'
    <IDENTIFIER> "1" , <ARRAY SEGMENT> "1"
    @ ? '(/1' <BOUND PAIR LIST> "1" /) %
==>
    <IDENTIFIER> "1" '(/1' <BOUND PAIR LIST> "1" /) ,
    <ARRAY SEGMENT> "1"
'SC'
%
```

```
'COMMENT' PRDECTRS VERSION6.  DENOTATION OF FORMAL AND VALUE PARAMETERS
    IN PROCEDURES, WITH INSERTION OF RESULT DECLARATIONS;

<PROCEDURE DECLARATION>
@

'COMMENT' 1, INSERTION OF RESULT DECLARATION, <RETURN STATEMENT>,
    AND DENOTATION OF RESULT IDENTIFIERS IN <TYPE> PROCEDURES;

'SD'
    <TYPE> "1" 'PROCEDURE' <PROCEDURE HEADING> "1"
    @ <IDENTIFIER> "1" ? % <STATEMENT> "1"
==>
    <PROCEDURE DECLARATION>
    @ <TYPE> "1" 'PROC' <PROCEDURE HEADING> "1"
        'BEGIN' 'DEC' <SERIAL NUMBER> "1" <TYPE> "1" 'RESULT';
            <STATEMENT> "1"
    @

    'COMMENT' 1.A.1.  DENOTE ASSIGNMENTS OF RESULT TO <TYPE> PROCEDURE
        IDENTIFIER;

    'SD'
        <STATEMENT> "11" @ ? <IDENTIFIER> "1" := ? %
    ==>
        <STATEMENT> "11"
        @ ? <IDENTIFIER>
            @ <SERIAL NUMBER> "1" <TYPE> "1" 'RESULT' % := ?
        %
    'SC'
    %
            ; 'RETURN' <SERIAL NUMBER> "1" <TYPE> "1" 'RESULT'
        'END'
    %
    @

    'COMMENT' 1.B.1.  DENOTATION OF ARRAYS CALLED BY VALUE;

    'SD'
        <TYPE> 'PROC' <IDENTIFIER> <FORMAL PARAMETER PART> "11" ;
        'VALUE' <IDENTIFIER LIST> "11"
        @ ? <IDENTIFIER> "11" @ <IDENTIFIER TOKEN> % ? % ;
        <SPECIFICATION PART> "11"
        @ ? <SPECIFIER> "11" @ ? 'ARRAY' % <IDENTIFIER LIST>
            @ ? <IDENTIFIER> "11" ? % ?
        % <BLOCK HEAD> "11" ; <COMPOUND TAIL> "11"
    ==>
        <TYPE> "1" 'PROC' <IDENTIFIER> "1" <FORMAL PARAMETER PART> "11";
        'VALUE' <IDENTIFIER LIST> "11"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "11" <SPECIFIER> "11" % ? % ;
        <SPECIFICATION PART> "11"
```

```
    <BLOCK HEAD> "11" ; 'DEC' <SERIAL NUMBER> "11" <SPECIFIER> "11"
        @ 'SD' 'ARRAY' ==> 'REAL' 'ARRAY' 'SC' % ;
        ALLOCATE ( <SERIAL NUMBER> "11" <SPECIFIER> "11"
        @ 'SD' 'ARRAY' ==> 'REAL' 'ARRAY' 'SC' % ,
            <IDENTIFIER> "11" ) ;
        <COMPOUND TAIL> "11"
    @

    'COMMENT' 1.B.1.A.1.  DENOTE OCCURRENCES OF THE VALUE ARRAY;

    'SD'
        <COMPOUND TAIL> "111" @ ? <IDENTIFIER> "11" ? %
    ==>
        <COMPOUND TAIL> "111"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "11" <SPECIFIER> "11"
           @ 'SD' 'ARRAY' ==> 'REAL' 'ARRAY' 'SC' % % ?
        %
    'SC'
    %
'SC'

'COMMENT' 1.B.2.  DENOTATION OF FORMAL PARAMETERS CALLED BY VALUE;

'SD'
    <TYPE> 'PROC' <IDENTIFIER> <FORMAL PARAMETER PART> "11" ;
    'VALUE' <IDENTIFIER LIST> "11"
    @ ? <IDENTIFIER> "11" @ <IDENTIFIER TOKEN> % ? % ;
    <SPECIFICATION PART> "11"
    @ ? <SPECIFIER> "11" <IDENTIFIER LIST>
        @ ? <IDENTIFIER> "11" ? % ?
    % <BLOCK HEAD> "11" ; <COMPOUND TAIL> "11"
==>
    <TYPE> "1" 'PROC' <IDENTIFIER> "1" <FORMAL PARAMETER PART> "11";
    'VALUE' <IDENTIFIER LIST> "11"
    @ ? <IDENTIFIER> @ <SERIAL NUMBER> "11" <SPECIFIER> "11" % ? % ;
    <SPECIFICATION PART> "11"
    <BLOCK HEAD> "11" ; 'DEC' <SERIAL NUMBER> "11" <SPECIFIER> "11";
        <SERIAL NUMBER> "11" <SPECIFIER> "11" := <IDENTIFIER> "11" ;
        <COMPOUND TAIL> "11"
    @

    'COMMENT' 1.B.2.A.1.  DENOTE OCCURRENCES OF THE VALUE PARAMETER;

    'SD'
        <COMPOUND TAIL> "111" @ ? <IDENTIFIER> "11" ? %
    ==>
        <COMPOUND TAIL> "111"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "11" <SPECIFIER> "11" % ? %
    'SC'
    %
'SC'
%
```

```
@

'COMMENT' 1.C.1.  DENOTATION OF FORMAL PARAMETERS;

'SD'
    <TYPE> 'PROC' <PROCEDURE HEADING> "21"
    @ <IDENTIFIER> ( <FORMAL PARAMETER LIST>
        @ <FORMAL PARAMETER> @ <IDENTIFIER> "21" %
            <PARAMETER DELIMITER> <FORMAL PARAMETER> "21"
        % ?
    % <BLOCK HEAD> "21" ; <COMPOUND TAIL> "21"
==>
    <TYPE> "1" 'PROC' <PROCEDURE HEADING> "21"
    @ <IDENTIFIER> "1" ( <FORMAL PARAMETER LIST>
        @ <FORMAL PARAMETER> "21" % ?
    % <BLOCK HEAD> "21" ; 'DEC' <SERIAL NUMBER> "21" 'FORMAL' ;
    <COMPOUND TAIL> "21"
    @

    'COMMENT' 1.C.1.A.1.  DENOTE OCCURRENCES OF FORMAL PARAMETER;

    'SD'
        <COMPOUND TAIL> "121" @ ? <IDENTIFIER> "21" ? %
    ==>
        <COMPOUND TAIL> "121"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "21" 'FORMAL' % ? %
    'SC'
    %
'SC'

'COMMENT' 1.C.2.  DENOTATION OF FORMAL PARAMETERS;

'SD'
    <TYPE> 'PROC' <PROCEDURE HEADING>
    @ <IDENTIFIER> ( <FORMAL PARAMETER> @ <IDENTIFIER> "21" % ) ? %
    <BLOCK HEAD> "21" ; <COMPOUND TAIL> "21"
==>
    <TYPE> "1" 'PROC' <IDENTIFIER> "1" ;
    <BLOCK HEAD> "21" ; 'DEC' <SERIAL NUMBER> "21" 'FORMAL' ;
    <COMPOUND TAIL> "21"
    @

    'COMMENT' 1.C.2.A.1.  DENOTE OCCURRENCES OF FORMAL PARAMETER;

    'SD'
        <COMPOUND TAIL> "121" @ ? <IDENTIFIER> "21" ? %
    ==>
        <COMPOUND TAIL> "121"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "21" 'FORMAL' % ? %
    'SC'
    %
'SC'
```

```
    %
'SC'

'COMMENT' 2, INSERTION OF RESULT DECLARATION IN NON-<TYPE> PROCEDURES;

'SD'
    'PROCEDURE' <PROCEDURE HEADING> "1" @ <IDENTIFIER> "1" ? %
    <STATEMENT> "1"
==>
    <PROCEDURE DECLARATION>
    @ 'PROC' <PROCEDURE HEADING> "1"
        'BEGIN' 'DEC' <SERIAL NUMBER> "1" 'RESULT';
            <STATEMENT> "1"
        'END'
    %
    @

    'COMMENT' 2.A.1.  DENOTATION OF ARRAYS CALLED BY VALUE;

    'SD'
        'PROC' <IDENTIFIER> <FORMAL PARAMETER PART> "11" ;
        'VALUE' <IDENTIFIER LIST> "11"
        @ ? <IDENTIFIER> "11" @ <IDENTIFIER TOKEN> % ? % ;
        <SPECIFICATION PART> "11"
        @ ? <SPECIFIER> "11" @ ? 'ARRAY' % <IDENTIFIER LIST>
            @ ? <IDENTIFIER> "11" ? % ?
        % <BLOCK HEAD> "11" ; <COMPOUND TAIL> "11"
    ==>
        'PROC' <IDENTIFIER> "1" <FORMAL PARAMETER PART> "11";
        'VALUE' <IDENTIFIER LIST> "11"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "11" <SPECIFIER> "11" % ? % ;
        <SPECIFICATION PART> "11"
        <BLOCK HEAD> "11" ; 'DEC' <SERIAL NUMBER> "11" <SPECIFIER> "11"
            @ 'SD' 'ARRAY' ==> 'REAL' 'ARRAY' 'SC' % ;
            ALLOCATE ( <SERIAL NUMBER> "11" <SPECIFIER> "11"
            @ 'SD' 'ARRAY' ==> 'REAL' 'ARRAY' 'SC' % ,
                <IDENTIFIER> "11" ) ;
            <COMPOUND TAIL> "11"
        @

        'COMMENT' 2.A.1.A.1.  DENOTE OCCURRENCES OF THE VALUE ARRAY;

        'SD'
            <COMPOUND TAIL> "111" @ ? <IDENTIFIER> "11" ? %
        ==>
            <COMPOUND TAIL> "111"
            @ ? <IDENTIFIER> @ <SERIAL NUMBER> "11" <SPECIFIER> "11"
               @ 'SD' 'ARRAY' ==> 'REAL' 'ARRAY' 'SC' % % ?
            %
        'SC'
        %
    'SC'
```

```
'COMMENT' 2.A.2.  DENOTATION OF FORMAL PARAMETERS CALLED BY VALUE;

'SD'
    'PROC' <IDENTIFIER> <FORMAL PARAMETER PART> "11" ;
    'VALUE' <IDENTIFIER LIST> "11"
    @ ? <IDENTIFIER> "11" @ <IDENTIFIER TOKEN> % ? % ;
    <SPECIFICATION PART> "11"
    @ ? <SPECIFIER> "11" <IDENTIFIER LIST>
        @ ? <IDENTIFIER> "11" ? % ?
    % <BLOCK HEAD> "11" ; <COMPOUND TAIL> "11"
==>
    'PROC' <IDENTIFIER> "1" <FORMAL PARAMETER PART> "11";
    'VALUE' <IDENTIFIER LIST> "11"
    @ ? <IDENTIFIER> @ <SERIAL NUMBER> "11" <SPECIFIER> "11" % ? % ;
    <SPECIFICATION PART> "11"
    <BLOCK HEAD> "11" ; 'DEC' <SERIAL NUMBER> "11" <SPECIFIER> "11";
        <SERIAL NUMBER> "11" <SPECIFIER> "11" := <IDENTIFIER> "11" ;
        <COMPOUND TAIL> "11"
    @

    'COMMENT' 2.A.2.A.1.  DENOTE OCCURRENCES OF THE VALUE PARAMETER;

    'SD'
        <COMPOUND TAIL> "111" @ ? <IDENTIFIER> "11" ? %
    ==>
        <COMPOUND TAIL> "111"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "11" <SPECIFIER> "11" % ? %
    'SC'
    %
'SC'
%
@

'COMMENT' 2.B.1.  DENOTATION OF FORMAL PARAMETERS;

'SD'
    'PROC' <PROCEDURE HEADING> "21"
    @ <IDENTIFIER> ( <FORMAL PARAMETER LIST>
        @ <FORMAL PARAMETER> @ <IDENTIFIER> "21" %
            <PARAMETER DELIMITER> <FORMAL PARAMETER> "21"
        % ?
    % <BLOCK HEAD> "21" ; <COMPOUND TAIL> "21"
==>
    'PROC' <PROCEDURE HEADING> "21"
    @ <IDENTIFIER> "1" ( <FORMAL PARAMETER LIST>
        @ <FORMAL PARAMETER> "21" % ?
    % <BLOCK HEAD> "21" ; 'DEC' <SERIAL NUMBER> "21" 'FORMAL' ;
    <COMPOUND TAIL> "21"
    @

    'COMMENT' 2.B.1.A.1.  DENOTE OCCURRENCES OF FORMAL PARAMETER;
```

```
        'SD'
            <COMPOUND TAIL> "121" @ ? <IDENTIFIER> "21" ? %
        ==>
            <COMPOUND TAIL> "121"
            @ ? <IDENTIFIER> @ <SERIAL NUMBER> "21" 'FORMAL' % ? %
        'SC'
        %
    'SC'

    'COMMENT' 2.B.2.  DENOTATION OF FORMAL PARAMETERS;

    'SD'
        'PROC' <PROCEDURE HEADING>
        @ <IDENTIFIER> ( <FORMAL PARAMETER> @ <IDENTIFIER> "21" % ) ? %
        <BLOCK HEAD> "21" ; <COMPOUND TAIL> "21"
    ==>
        'PROC' <IDENTIFIER> "1" ;
        <BLOCK HEAD> "21" ; 'DEC' <SERIAL NUMBER> "21" 'FORMAL' ;
        <COMPOUND TAIL> "21"
        @

        'COMMENT' 2.B.2.A.1.  DENOTE OCCURRENCES OF FORMAL PARAMETER;

        'SD'
            <COMPOUND TAIL> "121" @ ? <IDENTIFIER> "21" ? %
        ==>
            <COMPOUND TAIL> "121"
            @ ? <IDENTIFIER> @ <SERIAL NUMBER> "21" 'FORMAL' % ? %
        'SC'
        %
    'SC'
    %
'SC'
%
```

```
'COMMENT' BLOCKTRS VERSION10.  VERSNION7 TRANSFORMATIONS REWRITTEN
    TO PRESERVE ORDER OF IDENTIFIERS.  FOR USE WITH BLKHDTRS,
    ARSEGTRS, ARLSTTRS, PRDECTRS, AND FORSTTRS;

<BLOCK>
@

'COMMENT' 1.  DENOTATION OF SIMPLE VARIABLES;

'SD'
    <BLOCK HEAD> "1"
    @ ? <DECLARATION>
        @ <TYPE> "1" <IDENTIFIER> "1" @ <IDENTIFIER TOKEN> % % ?
    % ; <COMPOUND TAIL> "1"
==>
    <BLOCK>
    @ <BLOCK HEAD> "1"
        @ ? <DECLARATION> @ 'DEC' <SERIAL NUMBER> "1" <TYPE> "1" % ? % ;
        <COMPOUND TAIL> "1"
    %
    @

    'COMMENT' 1.A.1;

    'SD'
        <BLOCK> "11" @ ? <IDENTIFIER> "1" ? %
    ==>
        <BLOCK> "11"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "1" <TYPE> "1" % ? %
    'SC'
    %
'SC'

'COMMENT' 2.  DENOTATION OF ARRAYS;

'SD'
    <BLOCK HEAD> "1"
    @ ? <DECLARATION>
        @ <TYPE> "1" 'ARRAY' <IDENTIFIER> "1"
            '(/1' 1 : ALLOCATE ( <ACTUAL PARAMETER LIST> "1"
            @ <IDENTIFIER> ? % ) /)
        % ?
    % ;
    <COMPOUND TAIL> "1"
==>
    <BLOCK>
    @ <BLOCK HEAD> "1"
        @ ? <DECLARATION>
            @ 'DEC' <SERIAL NUMBER> "1" <TYPE> "1" 'ARRAY' % ?
        % ; ALLOCATE ( <ACTUAL PARAMETER LIST> "1"
        @ <IDENTIFIER> @ <SERIAL NUMBER> "1" <TYPE> "1" 'ARRAY' % ? % ) ;
```

```
        <COMPOUND TAIL> "1"
    %
    @

    'COMMENT' 2.A.1;

    'SD'
        <BLOCK> "11" @ ? <IDENTIFIER> "1" ? %
    ==>
        <BLOCK> "11"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "1" <TYPE> "1" 'ARRAY' % ? %
    'SC'
    %
'SC'

'COMMENT' 3.  DENOTATION OF LABELS, INCLUDING THOSE IN SWITCHES;

'SD'
    <BLOCK HEAD> "1" ; <COMPOUND TAIL> "1"
    @ ? <LABEL> @ <IDENTIFIER> "1" @ <IDENTIFIER TOKEN> % % : ? %
==>
    <BLOCK>
    @ <BLOCK HEAD> "1" ; 'DEC' <SERIAL NUMBER> "1" 'LABEL' ;
        <COMPOUND TAIL> "1"
    %
    @

    'COMMENT' 3.A.1;

    'SD'
        <BLOCK> "11" @ ? <IDENTIFIER> "1" ? %
    ==>
        <BLOCK> "11"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "1" 'LABEL' % ? %
    'SC'
    %
'SC'

'COMMENT' 4.  DENOTATION OF <TYPE> PROCEDURE IDENTIFIERS AND
    AND CREATION OF <SKIP STATEMENT>S;

'SD'
    <BLOCK HEAD> "1"
    @ ? <DECLARATION>
        @ <TYPE> "1" 'PROC' <IDENTIFIER> "1" ; <BLOCK> "1" % ?
    % ; <COMPOUND TAIL> "1"
==>
    <BLOCK>
    @ <BLOCK HEAD> "1"
        @ ? <DECLARATION>
            @ 'DEC' <SERIAL NUMBER> "1" <TYPE> "1" 'PROCEDURE' % ?
        % ;
```

```
        'SKIP' <SERIAL NUMBER> "1" <TYPE> "1" 'PROCEDURE' <BLOCK> "1" ;
        <COMPOUND TAIL> "1"
    %
    @

    'COMMENT' 4.A.1;

    'SD'
        <BLOCK> "11" @ ? <IDENTIFIER> "1" ? %
    ==>
        <BLOCK> "11"
        @ ? <IDENTIFIER>
            @ <SERIAL NUMBER> "1" <TYPE> "1" 'PROCEDURE' % ?
        %
    'SC'
    %
'SC'

'COMMENT' 5.  DENOTATION OF NON-<TYPE> PROCEDURE IDENTIFIERS AND
    AND CREATION OF <SKIP STATEMENT>S;

'SD'
    <BLOCK HEAD> "1"
    @ ? <DECLARATION> @ 'PROC' <IDENTIFIER> "1" ; <BLOCK> "1" % ? %
    ; <COMPOUND TAIL> "1"
==>
    <BLOCK>
    @ <BLOCK HEAD> "1"
        @ ? <DECLARATION>
            @ 'DEC' <SERIAL NUMBER> "1" 'PROCEDURE' % ?
        % ; 'SKIP' <SERIAL NUMBER> "1" 'PROCEDURE' <BLOCK> "1" ;
        <COMPOUND TAIL> "1"
    %
    @

    'COMMENT' 5.A.1;

    'SD'
        <BLOCK> "11" @ ? <IDENTIFIER> "1" ? %
    ==>
        <BLOCK> "11"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "1" 'PROCEDURE' % ? %
    'SC'
    %
'SC'

'COMMENT' 6.  DENOTATION OF SWITCH IDENTIFIERS;

'SD'
    <BLOCK HEAD> "1"
    @ ? <DECLARATION>
        @ 'SWITCH' <IDENTIFIER> "1" @ <IDENTIFIER TOKEN> %
```

```
            := <SWITCH LIST> "1"
        % ?
    % ; <COMPOUND TAIL> "1"
==>
    <BLOCK>
    @ <BLOCK HEAD> "1"
        @ ? <DECLARATION> @ 'DEC' <SERIAL NUMBER> "1" 'SWITCH' % ? % ;
        'SWITCH' <SERIAL NUMBER> "1" 'SWITCH' := <SWITCH LIST> "1" ;
        ALLOCATE ( <SERIAL NUMBER> "1" 'SWITCH' ) ; <COMPOUND TAIL> "1"
    %
    @

    'COMMENT' 6.A.1.  MOVE THE <SWITCH LIST> TO THE ALLOCATE PROCEDURE;

    'SD'
        <BLOCK HEAD> "11" ;
        'SWITCH' <IDENTIFIER> "11" := <SWITCH LIST> "11"
        @ <SWITCH LIST> @ <EXPRESSION> "11" , <EXPRESSION> "12" % ? % ;
        ALLOCATE ( <ACTUAL PARAMETER LIST> "11" ) ;
        <COMPOUND TAIL>
    ==>
        <BLOCK HEAD> "11" ;
        'SWITCH' <IDENTIFIER> "11" := <SWITCH LIST> "11"
        @ <SWITCH LIST> @ <EXPRESSION> "12" % ? % ;
        ALLOCATE ( <ACTUAL PARAMETER LIST> "11" , <EXPRESSION> "11" ) ;
        <COMPOUND TAIL> "1"
    'SC'

    'COMMENT' 6.A.2.  MOVE THE LAST ELEMENT OF THE <SWITCH LIST> AND
        DELETE THE OLD <SWITCH DECLARATION>;

    'SD'
        <BLOCK HEAD> "11" ; 'SWITCH' <IDENTIFIER> := <EXPRESSION> "11" ;
        ALLOCATE ( <ACTUAL PARAMETER LIST> "11" ) ; <COMPOUND TAIL>
    ==>
        <BLOCK HEAD> "11" ;
        ALLOCATE ( <ACTUAL PARAMETER LIST> "11" , <EXPRESSION> "11" ) ;
        <COMPOUND TAIL> "1"
    'SC'
    %
    @

    'COMMENT' 6.B.1;

    'SD'
        <BLOCK> "1" @ ? <IDENTIFIER> "1" ? %
    ==>
        <BLOCK> "1"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "1" 'SWITCH' % ? %
    'SC'
    %
'SC'
%
```

```
'COMMENT' FORSTTRS VERSION1.  TRANSFORMATION TO MAKE LABELS IN THE
    CONTROLLED STATEMENT OF A FOR STATEMENT LOCAL;

<FOR STATEMENT>
@

'COMMENT' 1.  DENOTE THE FIRST LABEL LOCAL TO THE CONTROLLED STATEMENT.
    (ANY OTHERS WILL BE DENOTED BY THE <BLOCK> TRANSFORMATIONS.);

'SD'
    <FOR CLAUSE> "1" <STATEMENT> "1"
    @ ? <LABEL> @ <IDENTIFIER> "1" @ <IDENTIFIER TOKEN> % % : ? %
==>
    <FOR CLAUSE> "1"
    'BEGIN' 'DEC' <SERIAL NUMBER> "1" 'LABEL' ;
        <STATEMENT> "1"
    @

    'COMMENT' 1.A.1.  DENOTE OCCURRENCES OF THE LABEL;

    'SD'
        <STATEMENT> "11" @ ? <IDENTIFIER> "1" ? %
    ==>
        <STATEMENT> "11"
        @ ? <IDENTIFIER> @ <SERIAL NUMBER> "1" 'LABEL' % ? %
    'SC'
    %
    'END'
'SC'
%
```

3.2 Transformations for For Statement Optimization

The for statement optimization problem in Algol 60 is broadly concerned with detecting and removing from the controlled statement of a for statement those calculations which can be guaranteed to be invariant or linear with respect to the loop, in the hope of reducing the overall execution time of the for statement. Since it is difficult to define in general what is the (execution-time) optimal form of a for statement*, much less to achieve it, the for statement optimization problem might more properly be called the "for statement improvement problem." Custom sanctions the former name, however. When optimization is restricted to subscript expressions, as is largely the case here, it is often called "recursive address calculation."

A number of papers discussing the general problem of program optimization have appeared in the literature since the publication of the classic paper of Samelson and Bauer [35] which introduces the concept of recursive address calculation. These include the discussions of for statement optimization in Hawkins and Huxtable [17], and in the work of the ALCOR group, Grau, Hill, and Langmaack [15], Bayer, *et al* [5], and Gries, *et al* [16]. Optimization of Fortran programs is discussed in three recent papers, those of Lowry and Medlock [22], Allen [1], and Busam and Englund [8]. Perforce the treatment of for statement optimization here is more closely

*For example, removal of invariant calculations may increase the execution time of a for statement if the numerical value of the until-expression is such that the controlled statement is executed zero times.

related to the concepts of the former group, especially [15] and [5], than to those of the latter.

The more recent papers above ([1] and [22], but also [17] to some extent) are concerned with performing extensive flow analysis on large segments of a program and attempting to optimize all suitable segments, not simply those associated with the particular syntactic notation provided for loops (e.g., the DO statement in Fortran). The descriptions of these processes tend to be quite complex and dependent on specific language constructions and array addressing schemes. The goal of the transformational description of for statement optimization presented here is rather different: It is to show that transformations can provide a reasonable and notationally convenient grammatical description of the (partially optimized) meaning of the syntactic construction for statement.

In the transformations presented here, attention is restricted to implementing recursive address calculation in for statements having a single step-until-element. (However, invariant step-and-until-expressions are also removed from the loop.) The reasons for this restriction are similar to those given in [15], namely that removal of other invariant or linear calculations is an editorial function that could as well be performed by the programmer. Hawkins and Huxtable [17] discuss the possibility of bringing other similar <for clause>s, e.g.:

for i := i+m while B do

under the optimization of step-until clauses; transformations could obviously be written to take care of such specific cases.

The details of the optimization process are discussed in the following subsection, while the for statement optimization transformation list itself is discussed in section 3.2.2.

3.2.1 Theory of For Statement Optimization

The optimization of a for statement may be divided into three subtasks: determination that the for statement is suitable for optimization (i.e., that invariant or linear calculations potentially can exist); determination that the subscripts of a particular subscripted variable in the controlled statement are linear or constant; and modification of the for statement to introduce recursive address calculation for such variables.

For purposes of the present discussion, a for statement is considered *suitable* for optimization if:

(1) The <for clause> consists of a single <for list element> of the step-until type.

(2) The loop variable is an integer simple variable (or integer formal parameter called by value).

(3) The loop variable is not assigned in the controlled statement.

(4) The step-expression does not contain the loop variable, nor the identifier of any variable assigned in the controlled statement.

(5) The for statement contains no procedure or function calls beyond the delimiter step (except for calls to standard functions, known not to cause side effects).

(6) The for statement contains no instances of a formal parameter called by name (except as an array identifier) beyond the delimiter step.

These conditions guarantee that the step-expression is constant with

respect to the loop, that adding it to the loop variable always produces the same increment in the loop variable, and that no side-effects can occur to cause hidden violations of this guarantee. (In addition to the above checks, the until-expression can be examined, and, if it does not contain the loop variable or a variable assigned in the controlled statement, it may be calculated before entering the loop.)

Subscripted variables occurring in the controlled statement of a for statement are classified as *constant*, *linear*, or *general* with respect to the loop according to the following criteria:

(1) A variable containing a variable assigned in the controlled statement is general.

(2) A variable declared in a block contained in the controlled statement is general.

(3) A variable containing a subscripted variable is general.

(4) Of the remaining variables, a variable not containing the loop variable is constant.

(5) Of the now remaining variables, a variable is linear provided each of its <subscript expression>s which contains the loop variable meets the following conditions:

(a) It contains only one instance of the loop variable.

(b) It does not contain: a real variable; a constant having an <exponent part> with negative exponent, or a <decimal fraction>; the operators /, // (integer division), or

'POWER'; a <conditional expression>; or a <procedure designator>.

(6) Any remaining variables are general.

Some of these criteria may be regarded as overly restrictive (for example, a[i+i] and a[i + cos(0)] are classed as general even though they actually are linear). In each of these cases what is essentially an engineering judgement has been made: that the frequency of occurrence of the particular case does not justify the increased complexity produced by refining the applicable criterion.

Recursive address calculation can be applied to the subscripted variables marked linear or constant according to the above criteria. To implement it, *reference primaries* and *address variables* are introduced, thereby avoiding explicit use of the addressing polynomial of an array. (This is similar to "Method I" of section 5.3.1 of [15].) The reasons for doing this are twofold: First, it keeps the transformational description of for statement optimization independent of the particular addressing scheme used for arrays (so long as that addressing scheme remains a linear function of the array subscripts). Second, it is appropriate for use with transformations because their lack of arithmetic capability makes it difficult to use them to reference the subscript range information contained in the "information vector" used with the addressing polynomial.

Reference primaries are defined by the production:

<primary> ::= ref <variable>

Their value is the address of <variable>. Address variables are defined by:

<variable> ::= val <simple variable>

They denote that a value is to be fetched from or stored into the address which is the value of <simple variable>. Using reference primaries and address variables, the base address and increment for a subscripted variable a[i] appearing in a statement controlled by the <for clause>:

for i := x step y until z do

are calculated as follows:

```
i := x;
base := ref a[i];
i := x+y;
increment := ref a[i]-base;
```

Occurrences of a[i] in the controlled statement are replaced by the address variable val base, and the statement:

base := base + increment

is placed at the end of the loop along with the statement incrementing i. (Appropriate declarations for the integer variables base and increment must also be introduced.) Subscripted variables constant with respect to the loop are treated similarly, but the variable increment and the calculations in which it appears are omitted.

The above method of recursive address calculation unfortunately becomes rather complicated when applied to nested for statements, because it is not possible to separate the dependence of the reference primaries on the various loop variables as can be done for the method of addressing polynomials. This problem is discussed at some length in section 5.4 of [15].

3.2.2 The For Statement Optimization Transformation Set

At the end of this section is listed the for statement optimization transformation set. It consists of but a single <statement>-transformation list which performs both the detection and optimization of suitable for statements and the expansion of the remainder in analogy with section 4.6 of the Revised Report. Extensive use is made of the ability of subtransformation sequences to localize the application of certain transformations, thereby avoiding pointless application to unsuitable statements.

The first transformation and the 1.A subtransformation list check the criteria for suitability given in section 3.2.1. Statements found to be unsuitable are marked with the marker forl, and are transformed later by transformations 2, 3, or 4. The 1.B subtransformation list then expands the <for clause> in a manner compatible with the 1.C subtransformation list and the later introduction of recursive address calculations. As a result of this expansion the until-expression is calculated before entering the loop. The 1.C subtransformation list examines the until-expression, and if it contains the loop variable or a variable assigned in the controlled statement it moves the calculation back into its proper place in the loop. In addition the 1.C subtransformation list moves and replicates the controlled statement to facilitate checking whether a subscripted variable contains a variable assigned in the loop. This check and the check for subscripted variables declared in blocks contained

in the controlled statement are performed by the 1.D subtransformation list. The 1.E subtransformation list then performs the remainder of the check to determine if a subscripted variable is constant, linear, or general with respect to the loop (cf. section 3.2.1).

The 1.F and 1.G subtransformation lists perform the actual introduction of recursive address calculation. The former optimizes subscripted variables marked linear, while the latter optimizes those marked constant. The notation, "use of <primary> is an artifact," in the comments preceding subtransformations 1.F.1.A.2 and 1.G.1.A.2 concerns an error in the grammar of section 6.1; address variables were introduced via the production:

<primary> ::= val <variable>

instead of:

<variable> ::= val <simple variable>

(Also, addr would probably be preferable to val)*. If the grammar were corrected, subtransformations 1.F.1.A.1 and 1.F.1.A.2 could be combined (as could 1.G.1.A.1 and 1.G.1.A.2). Subtransformations 1.F.1.A.3 and 1.G.1.A.3 apply when the variable to which recursive address calculation is being applied has already been optimized in an inner loop.

---

*Unfortunately, the computation of the tables required by the parsing algorithm from the grammar of section 6.1 required 3300 seconds on the IBM 360/75; therefore it was deemed impracticable to modify the grammar.

Transformations 2-8 of the <statement>-transformation list expand unsuitable for statements in a manner analogous to their definitions in section 4.6 of the Revised Report (see also the example at the end of section 1.1.2). The indexed symbol for "1" in these transformations matches either for or for1. Transformation 9 removes labels from a for statement and places them on a compound statement containing the for statement, so that transformations 1-8 can apply.

One should note that the for statement optimization transformation set presupposes the application of the identifier denotation transformation set described in section 3.1. This is necessary, for example, to permit the check that the loop variable of a suitable for statement is an integer simple variable, and to permit the check that a subscript containing the loop variable does not contain any real variables.

It should also be noted that this for statement optimization transformation set does not do an especially good job of optimizing nested for statements, since it does not attempt to move the recursive address calculation of a subscripted variable further out in the nest than the innermost loop at which the variable is linear. This problem requires further study to develop appropriate transformations.

The for statement optimization transformation set is listed below; examples of the application of the transformation set sequence consisting of the identifier denotation transformation set and the for statement optimization transformation set are given in section 6.3.

```
'COMMENT' FOROPTV7.  TRANSFORMATIONS TO CONVERT FOR STATEMENTS
    INTO STATEMENTS, OPTIMIZING THOSE WHICH ARE SUITABLE;

<STATEMENT>
@

'COMMENT' 1.  ANALYZE SIMPLE FOR-STEP-UNTIL STATEMENTS FOR
    OPTIMIZABILITY;

'SD'
    'FOR' <VARIABLE> "1" @ <SERIAL NUMBER> "1" 'INTEGER' % :=
    <EXPRESSION> "1" 'STEP' <EXPRESSION> "2" 'UNTIL'
    <EXPRESSION> "3" 'DO' <STATEMENT> "1"
==>
    <STATEMENT>
    @ 'FOR' <VARIABLE> "1" := <EXPRESSION> "1" 'STEP'
        <EXPRESSION> "2" 'UNTIL' <EXPRESSION> "3" 'DO'
        <STATEMENT> "1"
    %
    @

    'COMMENT' 1.A.1.  REMOVE UNARY PLUS FROM STEP EXPRESSION;

    'SD'
        'FOR' <VARIABLE> := <EXPRESSION> 'STEP' + <TERM> "1"
            'UNTIL' <EXPRESSION> 'DO'
        <STATEMENT>
    ==>
        'FOR' <VARIABLE> "1" := <EXPRESSION> "1" 'STEP' <TERM> "1"
            'UNTIL' <EXPRESSION> "3" 'DO'
        <STATEMENT> "1"
    'SC'

    'COMMENT' 1.A.2.  REMOVE FROM CONSIDERATION ANY STATEMENT IN
        WHICH SIDE EFFECTS COULD ARISE VIA PROCEDURE OR <TYPE>-
        PROCEDURE CALLS;

    'SD'
        <FOR STATEMENT> "11"
        @ 'FOR' <VARIABLE> := <EXPRESSION> ? <IDENTIFIER> "11"
            @ <SERIAL NUMBER> ? 'PROCEDURE' % ?
        %
    ==>
        <FOR STATEMENT> "11"
        @ 'FOR' @ 'FOR1' % <VARIABLE> "1" := <EXPRESSION> "1"
            ? <IDENTIFIER> "11" ?
        %
    'SC'

    'COMMENT' 1.A.3.  SIMILARLY FOR FORMAL PARAMETERS WHICH ARE
```

```
    USED AS SIMPLE VARIABLES;

'SD'
    <FOR STATEMENT> "11"
    @ 'FOR' <VARIABLE> := <EXPRESSION> ? <SIMPLE VARIABLE>  "11"
        @ <SERIAL NUMBER> 'FORMAL' % ?
    %
==>
    <FOR STATEMENT> "11"
    @ 'FOR' @ 'FOR1' % <VARIABLE> "1" := <EXPRESSION> "1"
        ? <SIMPLE VARIABLE> "11" ?
    %
'SC'

'COMMENT' 1.A.4.  SIMILARLY FOR FORMAL PARAMETERS WHICH ARE
    USED AS PROCEDURE IDENTIFIERS (CALLED <IDENTIFIER 1> BY
    SYNTAX);

'SD'
    <FOR STATEMENT> "11"
    @ 'FOR' <VARIABLE> := <EXPRESSION> ? <IDENTIFIER 1> "11"
        @ <SERIAL NUMBER> 'FORMAL' % ?
    %
==>
    <FOR STATEMENT> "11"
    @ 'FOR' @ 'FOR1' % <VARIABLE> "1" := <EXPRESSION> "1"
        ? <IDENTIFIER 1> "11" ?
    %
'SC'

'COMMENT' 1.A.5.  SIMILARLY FOR A STATEMENT IN WHICH THE LOOP
    VARIABLE IS ASSIGNED TO;

'SD'
    'FOR' <VARIABLE> := <FOR LIST> "11" 'DO' <STATEMENT>
    @ ? <VARIABLE> "1" := ? %
==>
    'FOR1' <VARIABLE> "1" := <FOR LIST> "11" 'DO'
        <STATEMENT> "1"
'SC'

'COMMENT' 1.A.6.  REMOVE FROM CONSIDERATION STATEMENTS IN
    WHICH THE STEP EXPRESSION CONTAINS THE LOOP VARIABLE;

'SD'
    'FOR' <VARIABLE> := <EXPRESSION> 'STEP'
    <EXPRESSION> @ ? <VARIABLE> "1" ? %
    'UNTIL' <EXPRESSION> 'DO' <STATEMENT>
==>
    'FOR1' <VARIABLE> "1" := <EXPRESSION> "1" 'STEP'
    <EXPRESSION> "2" 'UNTIL' <EXPRESSION> "3" 'DO'
    <STATEMENT> "1"
```

```
'SC'

'COMMENT' 1.A.7.  REMOVE FROM CONSIDERATION STATEMENTS IN
    WHICH THE STEP EXPRESSION CONTAINS A VARIABLE ASSIGNED
    INSIDE THE LOOP;

'SD'
    'FOR' <VARIABLE> := <EXPRESSION> 'STEP'
    <EXPRESSION> @ ? <IDENTIFIER> "11" ? %
    'UNTIL' <EXPRESSION> 'DO'
    <STATEMENT> @ ? <VARIABLE> @ <IDENTIFIER> "11" ? % := ? %
==>
    'FOR1' <VARIABLE> "1" := <EXPRESSION> "1" 'STEP'
    <EXPRESSION> "2" 'UNTIL' <EXPRESSION> "3" 'DO'
    <STATEMENT> "1"
'SC'
%
@

'COMMENT' 1.B.1.  EXPAND STEP-UNTIL CLAUSES IN ANALOGY WITH
    SECTION 4.6.4.2 OF THE REVISED REPORT, LEAVING PLACES
    TO INSERT ADDITIONAL STATEMENTS GENERATED IN THE COURSE
    OF OPTIMIZATION.  EXPAND THE CASE OF A POSITIVE INTEGER
    STEP FIRST;

'SD'
    'FOR' <VARIABLE> := <EXPRESSION> 'STEP'
        <UNSIGNED INTEGER> "11" 'UNTIL' <EXPRESSION> 'DO'
    <STATEMENT>
==>
    'BEGIN1' 'DEC' <SERIAL NUMBER> "11" 'LABEL' ;
        'DEC' <SERIAL NUMBER> "12" 'INTEGER' ;
        'BEGIN' <VARIABLE> "1" := <EXPRESSION> "1";
            <SERIAL NUMBER> "12" 'INTEGER' ..= <EXPRESSION> "3"
        'END' ;
        <STATEMENT> "1" ;
        <VARIABLE> "1" ..= <VARIABLE> "1" + <UNSIGNED INTEGER> "11" ;
        <VARIABLE> "1" ..= <VARIABLE> "1" - <UNSIGNED INTEGER> "11" ;
    <SERIAL NUMBER> "11" 'LABEL' :
        'IF' <VARIABLE> "1" <= <SERIAL NUMBER> "12" 'INTEGER' 'THEN'
        'BEGIN' <VARIABLE> "1" ..= <VARIABLE> "1"
                + <UNSIGNED INTEGER> "11" ;
            'GO TO' <SERIAL NUMBER> "11" 'LABEL'
        'END'
    'END'
'SC'

'COMMENT' 1.B.2.  EXPAND A NEGATIVE INTEGER STEP;

'SD'
    'FOR' <VARIABLE> := <EXPRESSION> 'STEP'
        - <UNSIGNED INTEGER> "11" 'UNTIL' <EXPRESSION> 'DO'
```

```
    <STATEMENT>
==>
    'BEGIN1' 'DEC' <SERIAL NUMBER> "11" 'LABEL' ;
        'DEC' <SERIAL NUMBER> "12" 'INTEGER' ;
        'BEGIN' <VARIABLE> "1" := <EXPRESSION> "1" ;
            <SERIAL NUMBER> "12" 'INTEGER' ..= <EXPRESSION> "3"
        'END' ;
        <STATEMENT> "1" ;
        <VARIABLE> "1" ..= <VARIABLE> "1" - <UNSIGNED INTEGER> "11" ;
        <VARIABLE> "1" ..= <VARIABLE> "1" + <UNSIGNED INTEGER> "11" ;
    <SERIAL NUMBER> "11" 'LABEL' :
        'IF' <VARIABLE> "1" >= <SERIAL NUMBER> "12" 'INTEGER' 'THEN'
        'BEGIN' <VARIABLE> "1" ..= <VARIABLE> "1"
                - <UNSIGNED INTEGER> "11" ;
            'GO TO' <SERIAL NUMBER> "11" 'LABEL'
        'END'
    'END'
'SC'

'COMMENT' 1.B.3.  EXPAND THE GENERAL STEP ELEMENT;

'SD'
    'FOR' <VARIABLE> := <EXPRESSION> 'STEP'
        <EXPRESSION> 'UNTIL' <EXPRESSION> 'DO'
    <STATEMENT>
==>
    'BEGIN1' 'DEC' <SERIAL NUMBER> "11" 'LABEL' ;
        'DEC' <SERIAL NUMBER> "12" 'INTEGER' ;
        'DEC' <SERIAL NUMBER> "13" 'INTEGER' ;
        'BEGIN' <VARIABLE> "1" := <EXPRESSION> "1";
            <SERIAL NUMBER> "12" 'INTEGER' ..= <EXPRESSION> "2";
            <SERIAL NUMBER> "13" 'INTEGER' ..= <EXPRESSION> "3"
        'END' ;
        <STATEMENT> "1" ;
        <VARIABLE> "1" ..= <VARIABLE> "1" +
            <SERIAL NUMBER> "12" 'INTEGER' ;
        <VARIABLE> "1" ..= <VARIABLE> "1" -
            <SERIAL NUMBER> "12" 'INTEGER' ;
    <SERIAL NUMBER> "11" 'LABEL' :
        'IF' ( <VARIABLE> "1" - <SERIAL NUMBER> "13" 'INTEGER' )
            * SIGN ( <SERIAL NUMBER> "12" 'INTEGER' ) <= 0
        'THEN'
        'BEGIN' <VARIABLE> "1" ..= <VARIABLE> "1"
                + <SERIAL NUMBER> "12" 'INTEGER' ;
            'GO TO' <SERIAL NUMBER> "11" 'LABEL'
        'END'
    'END'
'SC'
%
@

'COMMENT' 1.C.1.  IF UNTIL EXPRESSION CONTAINS LOOP VARIABLE
```

```
    AND SHOULD NOT BE CALCULATED IN ADVANCE, REMOVE SAID
    CALCULATION;

'SD'
    <BLOCK HEAD> "11" ; 'DEC' <SERIAL NUMBER> "11" 'INTEGER' ;
        'BEGIN' <COMPOUND TAIL> "11"
        @ ? <COMPOUND TAIL>
            @ <STATEMENT> "11" ; <SERIAL NUMBER> "11" 'INTEGER'
                ..= <EXPRESSION> @ ? <VARIABLE> "1" ? % 'END'
            %
        % ; <STATEMENT> ;
        <STATEMENT> "12" ;
        <STATEMENT> "13" ;
    <LABEL> "11" :
        'IF' <EXPRESSION> "11"
        @ ? <PRIMARY> @ <SERIAL NUMBER> "11" 'INTEGER' % ? % 'THEN'
        'BEGIN' <COMPOUND TAIL> "12"
    'END'
==>
    <BLOCK HEAD> "11" ;
        'BEGIN' <COMPOUND TAIL> "11"
        @ ? <COMPOUND TAIL> @ <STATEMENT> "11" 'END' % % ;
        'BEGIN' 'END' ; <STATEMENT> "12" ;
        'BEGIN' 'END' ; <STATEMENT> "13" ;
    <LABEL> "11" :
        'IF' <EXPRESSION> "11"
        @ ? <PRIMARY> @ ( <EXPRESSION> "3" ) % ? % 'THEN'
        'BEGIN'
            'BEGIN' <STATEMENT> "1" ; <STATEMENT> "1" 'END' ;
        <COMPOUND TAIL> "12"
    'END'
'SC'

'COMMENT' 1.C.2.  IF UNTIL EXPRESSION CONTAINS A VARIABLE
    ASSIGNED IN THE LOOP AND SHOULD NOT BE CALCULATED IN
    ADVANCE, REMOVE SAID CALCULATION;

'SD'
    <BLOCK HEAD> "11" ; 'DEC' <SERIAL NUMBER> "11" 'INTEGER' ;
        'BEGIN' <COMPOUND TAIL> "11"
        @ ? <COMPOUND TAIL>
            @ <STATEMENT> "11" ; <SERIAL NUMBER> "11" 'INTEGER'
                ..= <EXPRESSION>
                @ ? <VARIABLE> @ <IDENTIFIER> "11" ? % ? % 'END'
            %
        % ; <STATEMENT>
        @ ? <VARIABLE> @ <IDENTIFIER> "11" ? % := ? % ;
        <STATEMENT> "12" ;
        <STATEMENT> "13" ;
    <LABEL> "11" :
        'IF' <EXPRESSION> "11"
        @ ? <PRIMARY> @ <SERIAL NUMBER> "11" 'INTEGER' % ? % 'THEN'
```

```
        'BEGIN' <COMPOUND TAIL> "12"
    'END'
==>
    <BLOCK HEAD> "11" ;
        'BEGIN' <COMPOUND TAIL> "11"
        @ ? <COMPOUND TAIL> @ <STATEMENT> "11" 'END' % % ;
        'BEGIN' 'END' ; <STATEMENT> "12" ;
        'BEGIN' 'END' ; <STATEMENT> "13" ;
    <LABEL> "11" :
        'IF' <EXPRESSION> "11"
        @ ? <PRIMARY> @ ( <EXPRESSION> "3" ) % ? % 'THEN'
        'BEGIN'
            'BEGIN' <STATEMENT> "1" ; <STATEMENT> "1" 'END' ;
        <COMPOUND TAIL> "12"
    'END'
'SC'

'COMMENT' 1.C.3.  IF UNTIL EXPRESSION COULD BE CALCULATED IN
    ADVANCE, MOVE AND REPLICATE THE CONTROLLED STATEMENT AND
    INSERT DUMMY COMPOUND STATEMENTS AS REQUIRED;

'SD'
    <BLOCK HEAD> "11" ; <COMPOUND STATEMENT> "11" ;
        <STATEMENT> ; <STATEMENT> "11" ; <STATEMENT> "12" ;
    <LABEL> "11" :
        <IF CLAUSE> "11" 'BEGIN' <COMPOUND TAIL> "11"
   'END'
==>
    <BLOCK HEAD> "11" ; <COMPOUND STATEMENT> "11" ;
        'BEGIN' 'END' ; <STATEMENT> "11" ;
        'BEGIN' 'END' ; <STATEMENT> "12" ;
    <LABEL> "11" :
        <IF CLAUSE> "11"
        'BEGIN'
            'BEGIN' <STATEMENT> "1" ; <STATEMENT> "1" 'END' ;
        <COMPOUND TAIL> "11"
    'END'
'SC'
%
@

'COMMENT' 1.D.1.  MARK ALL SUBSCRIPTED VARIABLES IN THE
    CONTROLLED STATEMENT WHICH CONTAIN THE IDENTIFIERS OF
    VARIABLES ASSIGNED TO;

'SD'
    <BLOCK HEAD> "11" ; <STATEMENT> "11" ; <STATEMENT> "12" ;
        <STATEMENT> "13" ; <STATEMENT> "14" ; <STATEMENT> "15" ;
    <LABEL> "11" :
        <IF CLAUSE> "11"
        'BEGIN'
            'BEGIN' <STATEMENT> "16"
```

```
            @ ? <VARIABLE> "11" @ <IDENTIFIER> "11" ? % := ? % ;
                <STATEMENT> "17"
            'END' ;
        <COMPOUND TAIL> "11"
    'END'
==>
    <BLOCK HEAD> "11" ; <STATEMENT> "11" ; <STATEMENT> "12" ;
        <STATEMENT> "13" ; <STATEMENT> "14" ; <STATEMENT> "15" ;
    <LABEL> "11" :
        <IF CLAUSE> "11"
        'BEGIN'
            'BEGIN' <STATEMENT> "16"
            @ ? <VARIABLE> "11" ..= ? % ;
            <STATEMENT> "17"
    @

    'COMMENT' 1.D.1.A.1.  MARK THE VARIABLES;

    'SD'
        <STATEMENT> "111"
        @ ? <SUBSCRIPTED VARIABLE>
            @ <IDENTIFIER> "111" (/ <SUBSCRIPT LIST> "111"
                @ ? <IDENTIFIER> "11" ? % /)
            % ?
        %
    ==>
        <STATEMENT> "111"
        @ ? <SUBSCRIPTED VARIABLE>
            @ <IDENTIFIER> "111" '(/1' <SUBSCRIPT LIST> "111" /) % .
        %
    'SC'
    %
            'END' ;
        <COMPOUND TAIL> "11"
    'END'
'SC'

'COMMENT' 1.D.2.  REMOVE THE COPY OF THE CONTROLLED STATEMENT
    USED TO CHECK ASSIGNMENT, AND MARK SUBSCRIPTED VARIABLES
    DECLARED IN BLOCKS WITHIN THE CONTROLLED STATEMENT;

'SD'
    <BLOCK HEAD> "11" ; <STATEMENT> "11" ; <STATEMENT> "12" ;
        <STATEMENT> "13" ; <STATEMENT> "14" ; <STATEMENT> "15" ;
    <LABEL> "11" :
        <IF CLAUSE> "11"
        'BEGIN'
            'BEGIN' <STATEMENT> "16" ; <STATEMENT> "17" 'END' ;
         <COMPOUND TAIL> "11"
    'END'
==>
    <BLOCK HEAD> "11" ; <STATEMENT> "11" ; <STATEMENT> "12" ;
```

```
        <STATEMENT> "13" ; <STATEMENT> "14" ; <STATEMENT> "15" ;
    <LABEL> "11" :
        <IF CLAUSE> "11"
        'BEGIN2' <STATEMENT> "17"
    @

    'COMMENT' 1.D.2.A.1.  LOCATE SUBSCRIPTED VARIABLES
        DECLARED IN BLOCKS CONTAINED IN THE CONTROLLED
        STATEMENT;

    'SD'
        <STATEMENT> "111"
        @ ? <BLOCK HEAD> "111"
            @ 'BEGIN' ? 'DEC' <DENOTATION>
                @ <SERIAL NUMBER> "111" ? 'ARRAY' % ?
            % ; <COMPOUND TAIL> "111"
            @ ? <SUBSCRIPTED VARIABLE> "111"
                @ <DENOTATION> "111"
                    @ <SERIAL NUMBER> "111" ? % (/ ?
                % ?
            % ?
        %
    ==>
        <STATEMENT> "111"
        @ ? <BLOCK HEAD> "111" ; <COMPOUND TAIL> "111"
            @ ? <SUBSCRIPTED VARIABLE> "111"
                @ <DENOTATION> "111" '(/1' ? % ?
            % ?
        %
    'SC'
    % ; <COMPOUND TAIL> "11"
    'END'
'SC'
%
@

'COMMENT' 1.E.1.  MARK THE REMAINING UNMARKED SUBSCRIPTED
    VARIABLES AS GENERAL ( '(/1' ), LINEAR IN THE LOOP
    VARIABLE ( '(/2' ), OR CONSTANT ( '(/3' );

'SD'
    <BLOCK HEAD> "11" ; <STATEMENT> "11" ; <STATEMENT> "12" ;
        <STATEMENT> "13" ; <STATEMENT> "14" ; <STATEMENT> "15" ;
    <LABEL> "11" : <IF CLAUSE> "11"
        'BEGIN2' <STATEMENT> "16" ; <COMPOUND TAIL> "11"
    'END'
==>
    <BLOCK HEAD> "11" ; <STATEMENT> "11" ; <STATEMENT> "12" ;
        <STATEMENT> "13" ; <STATEMENT> "14" ; <STATEMENT> "15" ;
    <LABEL> "11" : <IF CLAUSE> "11"
        'BEGIN' <STATEMENT> "16"
    @
```

```
'COMMENT' 1.E.1.A.1.  CHECK A SUBSCRIPTED VARIABLE BY
    CHECKING THE EXPRESSIONS IN IT;

'SD'
    <STATEMENT> "111"
    @ ? <SUBSCRIPTED VARIABLE> "111"
        @ <IDENTIFIER> (/ ? <VARIABLE> "1" ? /) % ?
    %
==>
    <STATEMENT> "111"
    @ ? <SUBSCRIPTED VARIABLE> "111"
    @

    'COMMENT' 1.E.1.A.1.A.1.  IF THE SUBSCRIPTED VARIABLE
        CONTAINS ANOTHER SUBSCRIPTED VARIABLE, MARK IT
        GENERAL;

    'SD'
        <IDENTIFIER> "115" (/ <SUBSCRIPT LIST> "115"
        @ ? <SUBSCRIPTED VARIABLE> ? % /)
    ==>
        <IDENTIFIER> "115" '(/1' <SUBSCRIPT LIST> "115" /)
    'SC'
    %
    @

    'COMMENT' 1.E.1.A.1.B.1.  IF THE EXPRESSION BEING EXAMINED
        CONTAINS THE LOOP VARIABLE EXAMINE IT FOR LINEARITY;

    'SD'
        <IDENTIFIER> "115" (/ <SUBSCRIPT LIST> "115"
        @ ? <SUBSCRIPT EXPRESSION> "115"
            @ '(3' ? <VARIABLE> "1" ? % ?
        % /)
    ==>
        <IDENTIFIER> "115" (/ <SUBSCRIPT LIST> "115"
        @ ? <SUBSCRIPT EXPRESSION> "115"
        @

        'COMMENT' 1.E.1.A.1.B.1.A.1.  LOOK FOR TWO
            OCCURRENCES OF THE LOOP VARIABLE;

        'SD'
            '(3' <EXPRESSION> "115"
            @ ? <VARIABLE> "1" ? <VARIABLE> "1" ? % )
        ==>
            '(1' <EXPRESSION> "115" )
        'SC'

        'COMMENT' 1.E.1.A.1.B.1.A.2. LOOK FOR A REAL VARIABLE;
```

```
'SD'
    '(3' <EXPRESSION> "115"
    @ ? <IDENTIFIER> @ ? 'REAL' ? % ? % )
==>
    '(1' <EXPRESSION> "115" )
'SC'

'COMMENT' 1.E.1.A.1.B.1.A.3.  LOOK FOR A DECIMAL
    FRACTION;

'SD'
    '(3' <EXPRESSION> "115" @ ? <DECIMAL FRACTION> ? % )
==>
    '(1' <EXPRESSION> "115" )
'SC'

'COMMENT' 1.E.1.A.1.B.1.A.4.  LOOK FOR A NEGATIVE
    EXPONENT;

'SD'
    '(3' <EXPRESSION> "115"
    @ ? <EXPONENT PART> @ ' - <UNSIGNED INTEGER> % ? % )
==>
    '(1' <EXPRESSION> "115" )
'SC'

'COMMENT' 1.E.1.A.1.B.1.A.5.  LOOK FOR DIVISION;

'SD'
    '(3' <EXPRESSION> "115" @ ? / ? % )
==>
    '(1' <EXPRESSION> "115" )
'SC'

'COMMENT' 1.E.1.A.1.B.1.A.6.  LOOK FOR INTEGER DIVISION;

'SD'
    '(3' <EXPRESSION> "115" @ ? // ? % )
==>
    '(1' <EXPRESSION> "115" )
'SC'

'COMMENT' 1.E.1.A.1.B.1.A.7.  LOOK FOR EXPONENTIATION;

'SD'
    '(3' <EXPRESSION> "115" @ ? 'POWER' ? % )
==>
    '(1' <EXPRESSION> "115" )
'SC'

'COMMENT' 1.E.1.A.1.B.1.A.8.  LOOK FOR A CONDITIONAL
    EXPRESSION;
```

```
    'SD'
        '(3' <EXPRESSION> "115" @ ? <IF CLAUSE> ? % )
    ==>
        '(1' <EXPRESSION> "115" )
    'SC'

    'COMMENT' 1.E.1.A.1.B.1.A.9.  LOOK FOR A PROCEDURE
        DESIGNATOR;

    'SD'
        '(3' <EXPRESSION> "115"
        @ ? <PROCEDURE DESIGNATOR> ? % )
    ==>
        '(1' <EXPRESSION> "115" )
    'SC'

    'COMMENT' 1.E.1.A.1.B.1.A.10.  IF NONE OF THE ABOVE,
        MARK THE EXPRESSION LINEAR;

    'SD'
        '(3' <EXPRESSION> "115" )
    ==>
        '(2' <EXPRESSION> "115" )
    'SC'
    % ?
    % /)
'SC'

'COMMENT' 1.E.1.A.1.B.2.  IF THE EXPRESSION BEING
    EXAMINED DOES NOT CONTAIN THE LOOP VARIABLE,
    MARK IT LINEAR;

'SD'
    <IDENTIFIER> "115" (/ <SUBSCRIPT LIST> "115"
    @ ? <SUBSCRIPT EXPRESSION> "115" @ '(3' ? % ? % /)
==>
    <IDENTIFIER> "115" (/ <SUBSCRIPT LIST> "115"
    @ ? <SUBSCRIPT EXPRESSION> "115" @ '(2' ? % ? % /)
'SC'

'COMMENT' 1.E.1.A.1.B.3.  MARK NEXT EXPRESSION FOR ANALYSIS;

'SD'
    <IDENTIFIER> "115" (/ <SUBSCRIPT LIST> "115"
    @ ? <EXPRESSION> @ '(2' <EXPRESSION> "115" ) % ,
        <EXPRESSION> "116" ?
    % /)
==>
    <IDENTIFIER> "115" (/ <SUBSCRIPT LIST> "115"
    @ ? <EXPRESSION> "115" , <EXPRESSION>
        @ '(3' <EXPRESSION> "116" ) % ?
```

```
        % /)
    'SC'

    'COMMENT' 1.E.1.A.1.B.4.  IF THERE IS NO NEXT EXPRESSION,
        THE VARIABLE IS LINEAR;

    'SD'
        <IDENTIFIER> "115" (/ <SUBSCRIPT LIST> "115"
        @ ? <SUBSCRIPT EXPRESSION>
            @ '(2' <EXPRESSION> "115" ) %
        % /)
    ==>
        <IDENTIFIER> "115" '(/2' <SUBSCRIPT LIST> "115"
        @ ? <SUBSCRIPT EXPRESSION> @ <EXPRESSION> "115" % %
        /)
    'SC'

    'COMMENT' 1.E.1.A.1.B.5.  IF THERE IS A NONLINEAR SUBSCRIP
        EXPRESSION, MARK THE VARIABLE GENERAL;

    'SD'
        <IDENTIFIER> "115" (/ <SUBSCRIPT LIST> "115"
        @ ? <SUBSCRIPT EXPRESSION>
            @ '(1' <EXPRESSION> "115" ) % ?
        % /)
    ==>
        <IDENTIFIER> "115" '(/1' <SUBSCRIPT LIST> "115"
        @ ? <SUBSCRIPT EXPRESSION> @ <EXPRESSION> "115" % ? %
        /)
    'SC'

    'COMMENT' 1.E.1.A.1.B.6.  MARK THE FIRST SUBSCRIPT
        EXPRESSION FOR ANALYSIS;

    'SD'
        <IDENTIFIER> "115" (/ <SUBSCRIPT LIST> "115"
        @ <SUBSCRIPT EXPRESSION> @ <EXPRESSION> "115" % ? %
        /)
    ==>
        <IDENTIFIER> "115" (/ <SUBSCRIPT LIST> "115"
        @ <SUBSCRIPT EXPRESSION>
            @ '(3' <EXPRESSION> "115" ) % ?
        % /)
    'SC'
    % ?
    %
'SC'

'COMMENT' 1.E.1.A.2.  IF THE SUBSCRIPTED VARIABLE IS CONSTANT,
    MARK IT SO;

'SD'
```

```
        <STATEMENT> "111"
        @ ? <SUBSCRIPTED VARIABLE> "111"
            @ <IDENTIFIER> "111" (/ ? % ?
        %
    ==>
        <STATEMENT> "111"
        @ ? <SUBSCRIPTED VARIABLE> "111"
            @ <IDENTIFIER> "111" '(/3' ? % ?
        %
    'SC'
    % ; <COMPOUND TAIL> "11"
    'END'
'SC'
%
@

'COMMENT' 1.F.1.  DETECT AND OPTIMIZE SUBSCRIPTED VARIABLES WITH
    LINEAR SUBSCRIPTS;

'SD'
    <BLOCK HEAD> "11" ; <STATEMENT> "11" ;
        'BEGIN' <COMPOUND TAIL> "11" ; <STATEMENT> "12" ;
        'BEGIN' <COMPOUND TAIL> "12" ; <STATEMENT> "13" ;
    <LABEL> "11" : <IF CLAUSE> "11"
        'BEGIN' <STATEMENT> "14"
        @ ? <SUBSCRIPTED VARIABLE> "11"
            @ <IDENTIFIER> "11" '(/2' <SUBSCRIPT LIST> "11" /) % ?
        % ; <COMPOUND TAIL> "13"
    'END'
==>
    <BLOCK HEAD> "11" ; 'DEC' <SERIAL NUMBER> "11" 'INTEGER' ;
        'DEC' <SERIAL NUMBER> "12" 'INTEGER' ; <STATEMENT> "11" ;
        'BEGIN' <SERIAL NUMBER> "11" 'INTEGER' ..=
            'REF' <IDENTIFIER> "11" '(/1' <SUBSCRIPT LIST> "11" /) ;
        <COMPOUND TAIL> "11" ; <STATEMENT> "12" ;
        'BEGIN' <SERIAL NUMBER> "12" 'INTEGER' ..=
            'REF' <IDENTIFIER> "11" '(/1' <SUBSCRIPT LIST> "11" /)
            - <SERIAL NUMBER> "11" 'INTEGER' ;
        <COMPOUND TAIL> "12" ; <STATEMENT> "13" ;
    <LABEL> "11" : <IF CLAUSE> "11"
        'BEGIN' <STATEMENT> "14"
    @

    'COMMENT' 1.F.1.A.1.  REPLACE ALL OCCURRENCES OF THIS
        SUBSCRIPTED VARIABLE BY THE VALUE OF ITS REFERENCE;

    'SD'
        <STATEMENT> "111"
        @ ? <PRIMARY> @ <SUBSCRIPTED VARIABLE> "11" % ? %
    ==>
        <STATEMENT> "111"
        @ ? <PRIMARY> @ 'VAL' <SERIAL NUMBER> "11" 'INTEGER' % ? %
```

```
    'SC'

    'COMMENT' 1.F.1.A.2.  REPLACE SUBSCRIPTED VARIABLES ON THE
        LEFT OF THE ASSIGNMENT SYMBOL.  USE OF <PRIMARY> IS AN
        ARTIFACT;

    'SD'
        <STATEMENT> "111"
        @ ? <SUBSCRIPTED VARIABLE> "11" := ? %
    ==>
        <STATEMENT> "111"
        @ ? <PRIMARY> @ 'VAL' <SERIAL NUMBER> "11" 'INTEGER' % ..= ? %
    'SC'

    'COMMENT' 1.F.1.A.3.  REPLACE EACH OCCURRENCE OF 'REF'
        SUBSCRIPTED VARIABLE BY ITS REFERENCE SERIAL NUMBER;

    'SD'
        <STATEMENT> "111"
        @ ? <PRIMARY> @ 'REF' <SUBSCRIPTED VARIABLE> "11" % ? %
    ==>
        <STATEMENT> "111"
        @ ? <PRIMARY> @ <SERIAL NUMBER> "11" 'INTEGER' % ? %
    'SC'
    % ;
            <SERIAL NUMBER> "11" 'INTEGER' ..=
                <SERIAL NUMBER> "11" 'INTEGER'
                + <SERIAL NUMBER> "12" 'INTEGER' ;
        <COMPOUND TAIL> "13"
    'END'
'SC'
%
@

'COMMENT' 1.G.1.  DETECT AND OPTIMIZE THE SUBSCRIPTED VARIABLES
    WITH CONSTANT SUBSCRIPTS;

'SD'
    <BLOCK HEAD> "11" ; <STATEMENT> "11" ;
        'BEGIN' <COMPOUND TAIL> "11" ; <STATEMENT> "12" ;
        <STATEMENT> "13" ; <STATEMENT> "14" ;
    <LABEL> "11" : <IF CLAUSE> "11"
        'BEGIN' <STATEMENT> "15"
        @ ? <SUBSCRIPTED VARIABLE> "11"
            @ <IDENTIFIER> "11" '(/3' <SUBSCRIPT LIST> "11" /) % ?
        % ; <COMPOUND TAIL> "12"
    'END'
==>
    <BLOCK HEAD> "11" ; 'DEC' <SERIAL NUMBER> "11" 'INTEGER' ;
        <STATEMENT> "11" ;
        'BEGIN' <SERIAL NUMBER> "11" 'INTEGER' ..=
            'REF' <IDENTIFIER> "11" (/ <SUBSCRIPT LIST> "11" /) ;
```

```
            <COMPOUND TAIL> "11" ; <STATEMENT> "12" ;
            <STATEMENT> "13" ; <STATEMENT> "14" ;
        <LABEL> "11" : <IF CLAUSE> "11"
            'BEGIN' <STATEMENT> "15"
        @

        'COMMENT' 1.G.1.A.1.  REPLACE EACH OCCURRENCE OF THE
            SUBSCRIPTED VARIABLE BY ITS REFERENCE;

        'SD'
            <STATEMENT> "111"
            @ ? <PRIMARY> @ <SUBSCRIPTED VARIABLE> "11" % ? %
        ==>
            <STATEMENT> "111"
            @ ? <PRIMARY> @ 'VAL' <SERIAL NUMBER> "11" 'INTEGER' % ? %
        'SC'

        'COMMENT' 1.G.1.A.2.  REPLACE SUBSCRIPTED VARIABLES ON
            THE LEFT OF ASSIGNMENT SYMBOL.  USE OF <PRIMARY>
            IS AN ARTIFACT;

        'SD'
            <STATEMENT> "111"
            @ ? <SUBSCRIPTED VARIABLE> "11" := ? %
        ==>
            <STATEMENT> "111"
            @ ? <PRIMARY> @ 'VAL' <SERIAL NUMBER> "11" 'INTEGER' % ..= ? %
        'SC'

        'COMMENT' 1.G.1.A.3.  REPLACE EACH OCCURRENCE OF 'REF'
            SUBSCRIPTED VARIABLE BY ITS REFERENCE INTEGER;

        'SD'
            <STATEMENT> "111"
            @ ? <PRIMARY> @ 'REF' <SUBSCRIPTED VARIABLE> "11" % ? %
        ==>
            <STATEMENT> "111"
            @ ? <PRIMARY> @ <SERIAL NUMBER> "11" 'INTEGER' % ? %
        'SC'
        % ;
            <COMPOUND TAIL> "12"
        'END'
    'SC'
    %
'SC'

'COMMENT' 2.  APPLY THE DEFINITION OF SECTION 4.6.4.2 OF THE REVISED
    REPORT (SLIGHTLY MODIFIED) TO FOR-STEP-UNTIL STATEMENTS NOT
    OPTIMIZED.  HERE  'FOR' "1"  MATCHES EITHER 'FOR' OR 'FOR1'.
    FIRST TAKE CARE OF POSITIVE INTEGER STEPS;

'SD'
```

```
    'FOR' "1" <VARIABLE> "1" := <EXPRESSION> "1" 'STEP'
        <UNSIGNED INTEGER> "1" 'UNTIL' <EXPRESSION> "3" 'DO'
    <STATEMENT> "1"
==>
    'BEGIN' 'DEC' <SERIAL NUMBER> "1" 'LABEL' ;
        <VARIABLE> "1" := <EXPRESSION> "1" ;
    <SERIAL NUMBER> "1" 'LABEL' :
        'IF' <VARIABLE> "1" <= ( <EXPRESSION> "3" ) 'THEN'
        'BEGIN' <STATEMENT> "1" ;
            <VARIABLE> "1" := <VARIABLE> "1" + <UNSIGNED INTEGER> "1" ;
            'GO TO' <SERIAL NUMBER> "1" 'LABEL'
        'END'
    'END'
'SC'

'COMMENT' 3.  SAME FOR NEGATIVE INTEGER STEPS;

'SD'
    'FOR' "1" <VARIABLE> "1" := <EXPRESSION> "1" 'STEP'
        - <UNSIGNED INTEGER> "1" 'UNTIL' <EXPRESSION> "3" 'DO'
    <STATEMENT> "1"
==>
    'BEGIN' 'DEC' <SERIAL NUMBER> "1" 'LABEL' ;
        <VARIABLE> "1" := <EXPRESSION> "1" ;
    <SERIAL NUMBER> "1" 'LABEL' :
        'IF' <VARIABLE> "1" >= ( <EXPRESSION> "3" ) 'THEN'
        'BEGIN' <STATEMENT> "1" ;
            <VARIABLE> "1" := <VARIABLE> "1" - <UNSIGNED INTEGER> "1" ;
            'GO TO' <SERIAL NUMBER> "1" 'LABEL'
        'END'
    'END'
'SC'

'COMMENT' 4.  SAME FOR THE GENERAL STEP EXPRESSION;

'SD'
    'FOR' "1" <VARIABLE> "1" := <EXPRESSION> "1" 'STEP' <EXPRESSION> "2"
        'UNTIL' <EXPRESSION> "3" 'DO'
    <STATEMENT> "1"
==>
    'BEGIN' 'DEC' <SERIAL NUMBER> "1" 'LABEL' ;
        <VARIABLE> "1" := <EXPRESSION> "1" ;
    <SERIAL NUMBER> "1" 'LABEL' :
        'IF' ( <VARIABLE> "1" - ( <EXPRESSION> "3" ) )
            * SIGN ( <EXPRESSION> "2" ) <= 0
        'THEN'
        'BEGIN' <STATEMENT> "1" ;
            <VARIABLE> "1" := <VARIABLE> "1" + ( <EXPRESSION> "2" ) ;
            'GO TO' <SERIAL NUMBER> "1" 'LABEL'
        'END'
    'END'
'SC'
```

```
'COMMENT' 5.  APPLY THE DEFINITION OF SECTION 4.6.4.3 OF THE REVISED
    REPORT (SLIGHTLY MODIFIED) TO FOR-WHILE STATEMENTS;

'SD'
    'FOR' "1" <VARIABLE> "1" := <EXPRESSION> "1"
        'WHILE' <EXPRESSION> "2" 'DO'
    <STATEMENT> "1"
==>
    'BEGIN' 'DEC' <SERIAL NUMBER> "1" 'LABEL' ;
    <SERIAL NUMBER> "1" 'LABEL' :
        <VARIABLE> "1" := <EXPRESSION> "1" ;
        'IF' <EXPRESSION> "2" 'THEN'
        'BEGIN' <STATEMENT> "1" ;
            'GO TO' <SERIAL NUMBER> "1" 'LABEL'
        'END'
    'END'
'SC'

'COMMENT' 6.  EXPAND THE ARITHMETIC EXPRESSION ACCORDING TO SECTION
    4.6.4.1;

'SD'
    'FOR' "1" <VARIABLE> "1" := <EXPRESSION> "1" 'DO' <STATEMENT> "1"
==>
    'BEGIN' <VARIABLE> "1" := <EXPRESSION> "1" ;
        <STATEMENT> "1"
    'END'
'SC'

'COMMENT' 7.  EXPAND MULTIPLE FOR LIST ELEMENTS WHEN THE CONTROLLED
    STATEMENT IS ALREADY A PROCEDURE;

'SD'
    'FOR' "1" <VARIABLE> "1" := <FOR LIST> "1" ,
        <FOR LIST ELEMENT> "1" 'DO' <PROCEDURE STATEMENT> "1"
==>
    'BEGIN'
        'FOR1' <VARIABLE> "1" := <FOR LIST> "1" 'DO'
            <PROCEDURE STATEMENT> "1" ;
        'FOR1' <VARIABLE> "1" := <FOR LIST ELEMENT> "1" 'DO'
            <PROCEDURE STATEMENT> "1"
    'END'
'SC'

'COMMENT' 8.  EXPAND MULTIPLE FOR LIST ELEMENTS FOR OTHER STATEMENTS;

'SD'
    'FOR' "1" <VARIABLE> "1" := <FOR LIST> "1" ,
        <FOR LIST ELEMENT> "1" 'DO' <STATEMENT> "1"
==>
    'BEGIN' 'DEC' <SERIAL NUMBER> "1" 'PROCEDURE' ;
```

```
        'SKIP' <SERIAL NUMBER> "1" 'PROCEDURE'
        'BEGIN' 'DEC' <SERIAL NUMBER> "2" 'RESULT' ;
            <STATEMENT> "1"
        'END' ;
        'FOR1' <VARIABLE> "1" := <FOR LIST> "1" 'DO'
            <SERIAL NUMBER> "1" 'PROCEDURE' ;
        'FOR1' <VARIABLE> "1" := <FOR LIST ELEMENT> "1" 'DO'
            <SERIAL NUMBER> "1" 'PROCEDURE'
    'END'
'SC'

'COMMENT' 9.  RESTRUCTURE LABELED FOR STATEMENTS;

'SD'
    <LABEL> "1" : <FOR STATEMENT> "1"
==>
    <LABEL> "1" : 'BEGIN' <FOR STATEMENT> "1" 'END'
'SC'
%
```

4. Discussion and Conclusion

In this section I discuss some aspects of the intra-grammatical transformation system considered *in toto*. These include certain of the restrictions introduced in the consistency condition on IGTs and their relation to the overall system. In addition, some extensions to the system are considered; these range from fairly simple to rather fundamental, and indicate directions for further research. Additional potential applications of IGTs beyond those given here are also discussed; they, too, indicate areas for further investigation. Finally, some parallels between IGTs and natural language transformational grammars are briefly outlined.

4.1 Comments on the Definition of Intra-Grammatical Transformations

Three aspects of the definition of IGTs require further brief discussion; these are the intra-grammaticality restriction, the restriction that the variable of an indefinite pattern occur uniquely in the SDs of an IGT, and the difference between the sequencing rules for transformations and subtransformations.

The restriction that the result of applying a transformation to a parse tree be a parse tree of the same grammar as that of the parse tree transformed has two important consequences: first, it avoids the problem of specifying the tree structure for the transformed parse tree and permits assignment of a tree structure to the transformation itself; and second, it permits the tree structure of indefinites to be derived from their match in the SD, thereby avoiding the necessity of reparsing the transformed string after each application of a transformation. Such reparsing would, of course, considerably increase the computation required to apply a transformation set sequence to a program.

In this context one should also note that the manner in which the grammar used is extended to cover the constructions introduced via transformations can greatly affect their elegance and simplicity. For example, the transformations for identifier denotation would be considerably more complex than those of section 3.1.2 if <denotation>s had been introduced into the grammar by *adding* productions with <denotation> substituted for <identifier> rather than

making identifiers <identifier token>s and adding the single production:

<identifier> ::= <identifier token> | <denotation>

Also, in certain cases appropriate choice of left or right recursive productions for certain constructions in the grammar can simplify the transformations. Thus the Algol 60 productions for <block head> and <compound tail> are particularly convenient in the transformations which move information from the array, switch, and procedure declarations of a block into its <compound tail>.

The preparation of the formal definition of IGTs given in section 2 uncovered certain anomalies in the implementation of IGTs. One of these was the restriction that the occurrence of the variable of an indexed indefinite pattern be unique in the SDs of an IGT. This restriction is necessary to insure that an occurrence of that variable in an indefinite pattern in the SC denotes an element of the environment having a unique indefinite skeleton. The formal definition shows that a variable can actually stand for two different things: the identity of a particular parse tree, or the indefinite skeleton associated with a particular match of that parse tree. It might, therefore, be appropriate to have separate indices for parse trees and indefinite skeletons, but how to do so while retaining the simplicity of the present notation is a difficult problem. Permitting indices on free symbols might be one solution (as was in fact done in an early version

of IGTs), but doing so suggests that the free symbols can be rearranged, which, of course, they cannot without destroying the tree structure they represent.

The differences in the sequencing rules for transformations and subtransformations, specifically the fact that a transformation set is reapplied throughout a transformed parse tree whereas the subtransformation lists of a subtransformation sequence are applied only at the top of the transformed subtree produced from their associated pattern tree, may seem at first to be another *ad hoc* consequence of the implementation of IGTs. The present rules certainly work well, however, and may be at least partially justified by the difference in scope rules for transformations and subtransformations; that is, since subtransformations have access to "global" variables not in their own SD, it may be reasonable to restrict them from indiscriminate application throughout the transformed subtree.

4.2 Possible Extensions to Intra-Grammatical Transformations

Several directions for extension of IGTs are possible. One simple and useful notational extension would be to introduce Markov *terminal* transformations in addition to the present *simple* transformations (cf. [29], cha. 5). A terminal transformation, if it applies, specifies that the application of the transformation list in which it appears is to be unconditionally terminated after applying it. Such transformations would be especially useful where it is presently necessary to place a marker or otherwise artifically change the transformed parse tree to terminate application of a transformation list (cf, for example, the introduction of the marker '(/1' in transformation 1 of the <array segment>-transformation list of section 3.1.2). For similar reasons, nonterminal markers (synonyms for nonterminal symbols) might be useful in addition to the presently available terminal markers.

Two other essentially notational extensions, somewhat more complex than the above, are the introduction of logical combinations of conditions for matching in the SD (including conjunction, disjunction, and negation of pattern trees at a particular node), and the introduction of a (parameterless) procedure mechanism for labeling groups of transformations. The former would be useful for combining certain transformations which have almost identical structure but now must be written separately (for example the 1.E.1.A.1.B.1.A subtransformation list of section 3.2.2). The latter extension would

be useful where the same subtransformation list or sequence appears in more than one transformation, but it would greatly complicate the scope rules for subtransformations.

In considering the above extensions one should keep in mind that, in view of the result of section 2.4, they would only increase the convenience of using IGTs, not their descriptive power. Thus their introduction must be weighed carefully against the notational problems it would cause, lest whatever elegance and convenience the present formulation possesses be sacrificed for no real gain.

Several aspects of the definition of IGTs are deserving of further experimental or theoretical investigation. Experimentation with variations of the sequencing rules employed here would be interesting. For example, instead of alternating the reapplication of the transformation set with the application of subtransformation sequences in the SC, all of the subtransformation sequences might be applied at their respective nodes and then the entire transformed parse tree rescanned (cf. 2.3.2(3) Trset Recursive Result) to reapply the transformation set. As there are no obvious theoretical criteria for selecting one or the other of these sequencing methods, experimentation could be used to verify which is more elegant or efficient.

Areas for theoretical investigation include attempting to derive conditions which guarantee termination of the application of a transformation list, and conditions which guarantee that the program after transformation is semantically equivalent to the program before transformation. Iturriaga [20] has made some progress toward solving

the former problem, but the presence of indefinites in IGTs complicates it considerably.

Lastly, it would be interesting to investigate the problem of adapting transformations to transform the *abstract* syntax of a program (in the sense of the Vienna Report) rather than its concrete syntax, as is done here.

4.3 Other Applications of Intra-Grammatical Transformations

A number of applications of transformations in addition to those discussed in section 3 seem worthy of investigation. One of these, a problem currently of interest at Argonne National Laboratory [12], is that of automatically or semiautomatically converting an existing program to use multiple-precision arithmetic. Involved would be conversion of all arithmetic operations into calls on subroutines or functions in a multiple-precision arithmetic package, and introduction of the appropriate array declarations for the multiple-precision variables. Assuming a suitable grammar for the language used, this problem should be quite amenable to transformational solution.

There is also widespread general interest in designing languages which can be extended by the programmer. One approach to this problem is to introduce a *macro facility* into higher-level programming languages. Macro expansion should be a natural application for transformations (and indeed first motivated this investigation of them), assuming the problem of conveniently extending the context free grammar of the language can be solved.

Finally, the application of transformations to other programming language definitional problems could be investigated. One of these is the description of the syntax of DO-loops in Fortran; it would be interesting to compare a transformational description with the rather inelegant extended-BNF description of Rabinowitz [32]. Two

other definitional problems potentially amenable to transformational solution are the *context condition* and the *extensions* of Algol 68 [36]. These problems are somewhat similar to the identifier denotation and for statement optimization problems of Algol 60. Some modifications might have to be made to IGTs in order to work with the infinite grammar of Algol 68.

4.4 Conclusion

This work was begun with the intent of showing that Chomsky's concept of transformational grammar could be adapted and implemented for use with programming languages, and that transformational grammars so adapted are useful for solving certain programming language definitional problems. I believe that all three of these goals have been achieved, although much further work could be done. There remain three related observations on the philosophical implications of this work.

The first concerns a hypothesis of J. C. Reynolds [33], somewhat analogous to the "deep structure hypothesis" of Chomsky [10], concerning programming languages, viz: When a programmer writes a program, he actually conceives of a completely unambiguous tree-like structure (in which there are no duplicated identifier names, all identifiers are directly associated with their types, etc.), which he then transcribes into a linear string in some programming language.

The role of the syntactic parts of the programming language definition (including the production rules, identifier scope rules, and "abbreviations" such as _for_ statements) is thus to provide the programmer a maximum of convenience in making that transcription by relieving him of having to concern himself unnecessarily with details such as duplication of identifier names and the construction of loops.

The ability to give a purely grammatical description of identifier denotation and the semantics of for statements, which may be regarded as helping to restore the linear string to the tree-like structure conceived by the programmer, thus lends support to this hypothesis.

These examples also support an observation by Chomsky [10] (also related to the deep structure hypothesis) that nearly all transformations proposed for use in natural language grammars have the property that the semantics of a sentence is invariant under transformation. He suggests that such invariance may be a *linguistic universal*, i.e., a property of all correct transformational grammars for all natural languages.

Even if the transformations in the examples of section 3 are regarded as *defining* identifier denotation and the semantics of for statements, these definitions must concur with our previous understanding of those concepts. Thus this work lends even further weight to the semantic invariance hypothesis.

Finally, the intra-grammatical transformation system suggests the hypothesis that bottom-up application of transformation sets is a linguistic universal. Such an application rule is suggested by Chomsky in [10] for natural languages, and by Grau, Hill, and Langmaak in [15] for the optimization of multiple for statements. And it is certainly supported by the examples of section 3, especially the identifier denotation transformation set.

5. Bibliography

1. Allen, F. E. Program optimization. In Halpern, M. I., and Shaw, C. J., (Eds.). *Annual Review in Automatic Programming*, 5. Pergamon, Oxford, 1969, pp. 239-307.

2. Bach, E. *An Introduction to Transformational Grammars*. Holt, Rinehart, and Winston, New York, 1964.

3. de Bakker, J. W. Semantics of programming languages. In Tou, J. T., (Ed.). *Advances in Information Systems Science*. Plenum Press, New York, 1969, pp. 173-227.

4. de Bakker, J. W. *Formal Definition of Programming Languages, with an Application to the Definition of ALGOL 60*. Mathematisch Centrum, Amsterdam, 1967.

5. Bayer, R., Murphree, E., Jr., and Gries, D. *User's Manual for the ALCOR-ILLINOIS-7090 ALGOL-60 Translator*. University of Illinois, Urbana, Ill., September, 1964.

6. Boyle, J. M. and Grau, A. A. An algorithmic semantics for ALGOL 60 identifier denotation. *Jour. ACM*, 17, 2 (April, 1970), 361-382.

7. Brody, T. A. *Symbol-Manipulation Techniques for Physics*. Gordon and Breach, New York, 1968.

8. Busam, V. A., and Englund, D. E. Optimization of expressions in Fortran. *Comm. ACM* 12, 12 (Dec. 1969), 666-674.

9. Caracciolo di Forino, A. Generalized Markov algorithms and automata. In Caianiello, E. R. (Ed.). *Automata Theory*. Academic Press, New York, 1966.

10. Chomsky, N. *Aspects of the Theory of Syntax*. M.I.T. Press, Cambridge, Massachusetts, 1965.

11. Chomsky, N. Three models for the description of language. *IRE Trans. Inform. Theor. IT-2* (1956), 113-124. Reprinted, with corrections, in Luce, R. D., Bush, R., and Galanter, E. (Eds.). *Readings in Mathematical Psychology, Vol. II*. Wiley, New York, 1965.

12. Cody, W. J., Jr. Private communication.

13. Davis, M. *Computability and Unsolvability*. McGraw-Hill, New York, 1958.

14. Friedman, J. A computer system for transformational grammar. *Comm. ACM* 12, 6 (June 1969), 341-348.

15. Grau, A. A., Hill, U., and Langmaack, H. *Translation of ALGOL 60*. Springer-Verlag, Berlin, 1967.

16. Gries, D., Paul, M., and Wiehle, H. R. Some techniques used in the ALCOR ILLINOIS 7090. *Comm. ACM* 8, 8 (August, 1965), 496-500.

17. Hawkins, E. N., and Huxtable, D. H. R. A multi-pass translation scheme for ALGOL 60. In Goodman, R., (Ed.). *Annual Review in Automatic Programming*, 3. MacMillan, New York, 1963, pp. 163-205.

18. Hays, D. G. *Introduction to Computational Linguistics*. American Elsevier, New York, 1967.

19. Ingerman, P. Z., and Merner, J. N. Suggestions on ALGOL 60 (Rome) issues. *Comm. ACM* 6, 1 (Jan. 1963), 20-23.

20. Iturriaga, R. Contributions to mechanical mathematics. Ph.D. Thesis, Carnegie-Mellon Univ., Pittsburgh, Penn., May, 1967.

21. Knuth, D. E. The remaining trouble spots in ALGOL 60. *Comm. ACM* 10, 10 (Oct. 1967), 611-618.

22. Lowry, E. S., and Medlock, C. W. Object code optimization. *Comm. ACM* 12, 1 (Jan. 1969), 13-22.

23. Lucas, P., Lauer, P., and Stigleitner, H. Method and notation for the formal definition of programming languages. IBM Laboratory Vienna, Techn. Report TR 25.087, 28 June, 1968.

24. Maruyama, K. A tree transformation system based on NUCLEOL. M.S. Thesis, Univ. of Ill., Urbana, Ill., Feb., 1970.

25. McCarthy, J. A formal description of a subset of ALGOL. In Steel, T. B., Jr. (Ed.). *Formal Language Description Languages for Computer Programming*. North-Holland, Amsterdam, 1966, 1-12.

26. McCarthy, J. Towards a mathematical science of computation. In Popplewell, C. M., (Ed.). *Information Processing 1962*. North-Holland, Amsterdam, 1963, 21-28.

27. McCarthy, J., *et al*. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Mass., 1968.

28. McKeeman, W. M., Horning, J. J., Nelson, E. C., and Wortman, D. B. The XPL compiler generator system. *Proc. 1968 FJCC*, Thompson, Washington, 1968, pp. 617-635.

29. Mendelson, E. *Introduction to Mathematical Logic*. D. Man Nostrand, Princeton, New Jersey, 1964.

30. Naur, P. (Ed.). Revised report on the algorithmic language ALGOL 60. *Comm. ACM* 6, 1 (Jan. 1963), 1-17.

31. Petrick, S. R. A recognition procedure for transformational grammars. Ph.D. Thesis, MIT, Cambridge, Mass., 1965.

32. Rabinowitz, I. N. Report on the algorithmic language FORTRAN II. *Comm. ACM* 5, 6 (June 1962), 327-337.

33. Reynolds, J. C. Private communication.

34. Reynolds, J. C. *COGENT Programming Manual*. Report ANL-7022, Argonne National Laboratory, Argonne, Ill., March 1965.

35. Samelson, K., and Bauer, F. L. Sequential formula translation. *Comm. ACM* 3, 2 (Feb. 1960), 76-83.

36. van Wijngaarden, A. (Ed.), Mailloux, B. J., Peck, J. E. L., and Koster, C. H. A. *Report on the Algorithmic Language Algol 68*. Report MR101, Mathematisch Centrum, Amsterdam, Feb. 1969.

6. Appendices

6.1 Grammar for Transformational Examples

Listed below is the modified Algol 60 grammar (and the synonyms for terminal symbols) used for the transformations and examples of sections 3, 6.2, and 6.3. The modifications to the Algol 60 grammar consist primarily in making <identifier> (actually <identifier token>), <unsigned integer>, <logical value>, <adding operator>, <multiplying operator>, <relational operator>, and <dummy statement> terminal symbols, and removing certain inconsistencies associated with <number>, <string>, <expression> (the distinction between arithmetic, Boolean, and designational expressions, which can only be made on the basis of type information for variables, is dropped), <identifier>, <function designator> (a function designator with no parameters is called a <variable>), <switch designator>, and <procedure designator>. In addition systematic changes were made to <procedure heading> to eliminate occurrences of the symbol <empty>. Finally, <unsigned integer> as a <label> is not allowed.

```
<INTEGER> ::= <UNSIGNED INTEGER>
<INTEGER> ::= <ADDING OPERATOR> <UNSIGNED INTEGER>
<DECIMAL FRACTION> ::= . <UNSIGNED INTEGER>
<EXPONENT PART> ::= ' <INTEGER>
<DECIMAL NUMBER> ::= <UNSIGNED INTEGER>
<DECIMAL NUMBER> ::= <DECIMAL FRACTION>
<DECIMAL NUMBER> ::= <UNSIGNED INTEGER> <DECIMAL FRACTION>
<UNSIGNED NUMBER> ::= <DECIMAL NUMBER>
<UNSIGNED NUMBER> ::= <EXPONENT PART>
<UNSIGNED NUMBER> ::= <DECIMAL NUMBER> <EXPONENT PART>
<STRING> ::= '(' ')'
<STRING> ::= '' ''
<IDENTIFIER> ::= <IDENTIFIER TOKEN>
<IDENTIFIER> ::= <DENOTATION>
<SIMPLE VARIABLE> ::= <IDENTIFIER>
<SUBSCRIPT EXPRESSION> ::= <EXPRESSION>
<SUBSCRIPT LIST> ::= <SUBSCRIPT EXPRESSION>
<SUBSCRIPT LIST> ::= <SUBSCRIPT LIST> , <SUBSCRIPT EXPRESSION>
<SUBSCRIPTED VARIABLE> ::= <IDENTIFIER> ( <SUBSCRIPT LIST> )
<VARIABLE> ::= <SIMPLE VARIABLE>
<VARIABLE> ::= <SUBSCRIPTED VARIABLE>
<PRIMARY> ::= <UNSIGNED NUMBER>
<PRIMARY> ::= <VARIABLE>
<PRIMARY> ::= <PROCEDURE DESIGNATOR>
<PRIMARY> ::= ( <EXPRESSION> )
<PRIMARY> ::= 'VAL' <VARIABLE>
<PRIMARY> ::= 'REF' <VARIABLE>
<FACTOR> ::= <PRIMARY>
<FACTOR> ::= <FACTOR> 'POWER' <PRIMARY>
<TERM> ::= <FACTOR>
<TERM> ::= <TERM> <MULTIPLYING OPERATOR> <FACTOR>
<ARITHMETIC EXPRESSION> ::= <TERM>
<ARITHMETIC EXPRESSION> ::= <ADDING OPERATOR> <TERM>
<ARITHMETIC EXPRESSION> ::= <ARITHMETIC EXPRESSION>
     <ADDING OPERATOR> <TERM>
<RELATION> ::= <ARITHMETIC EXPRESSION> <RELATIONAL OPERATOR>
     <ARITHMETIC EXPRESSION>
<BOOLEAN PRIMARY> ::= <LOGICAL VALUE>
<BOOLEAN PRIMARY> ::= <RELATION>
<BOOLEAN PRIMARY> ::= <ARITHMETIC EXPRESSION>
<BOOLEAN SECONDARY> ::= <BOOLEAN PRIMARY>
<BOOLEAN SECONDARY> ::= ¬ <BOOLEAN PRIMARY>
<BOOLEAN FACTOR> ::= <BOOLEAN SECONDARY>
<BOOLEAN FACTOR> ::= <BOOLEAN FACTOR> & <BOOLEAN SECONDARY>
<BOOLEAN TERM> ::= <BOOLEAN FACTOR>
<BOOLEAN TERM> ::= <BOOLEAN TERM> | <BOOLEAN FACTOR>
<IMPLICATION> ::= <BOOLEAN TERM>
<IMPLICATION> ::= <IMPLICATION> 'IMPLIES' <BOOLEAN TERM>
<SIMPLE BOOLEAN> ::= <IMPLICATION>
<SIMPLE BOOLEAN> ::= <SIMPLE BOOLEAN> 'EQUIV' <IMPLICATION>
<EXP IF CLAUSE> ::= <IF CLAUSE>
```

```
<EXPRESSION> ::= <SIMPLE BOOLEAN>
<EXPRESSION> ::= <EXP IF CLAUSE> <SIMPLE BOOLEAN>
     'ELSE' <EXPRESSION>
<LABEL> ::= <IDENTIFIER>
<UNLABELLED BASIC STATEMENT> ::= <ASSIGNMENT STATEMENT>
<UNLABELLED BASIC STATEMENT> ::= <GO TO STATEMENT>
<UNLABELLED BASIC STATEMENT> ::= <DUMMY STATEMENT>
<UNLABELLED BASIC STATEMENT> ::= <PROCEDURE STATEMENT>
<UNLABELLED BASIC STATEMENT> ::= <RETURN STATEMENT>
<UNLABELLED BASIC STATEMENT> ::= <SKIP STATEMENT>
<BASIC STATEMENT> ::= <UNLABELLED BASIC STATEMENT>
<BASIC STATEMENT> ::= <LABEL> : <BASIC STATEMENT>
<UNCONDITIONAL STATEMENT> ::= <BASIC STATEMENT>
<UNCONDITIONAL STATEMENT> ::= <COMPOUND STATEMENT>
<UNCONDITIONAL STATEMENT> ::= <BLOCK>
<STATEMENT> ::= <UNCONDITIONAL STATEMENT>
<STATEMENT> ::= <CONDITIONAL STATEMENT>
<STATEMENT> ::= <FOR STATEMENT>
<COMPOUND TAIL> ::= <STATEMENT> 'END'
<COMPOUND TAIL> ::= <STATEMENT> ; <COMPOUND TAIL>
<BLOCK HEAD> ::= 'BEGIN' <DECLARATION>
<BLOCK HEAD> ::= <BLOCK HEAD> ; <DECLARATION>
<UNLABELLED COMPOUND> ::= 'BEGIN' <COMPOUND TAIL>
<UNLABELLED BLOCK> ::= <BLOCK HEAD> ; <COMPOUND TAIL>
<COMPOUND STATEMENT> ::= <UNLABELLED COMPOUND>
<COMPOUND STATEMENT> ::= <LABEL> : <COMPOUND STATEMENT>
<BLOCK> ::= <UNLABELLED BLOCK>
<BLOCK> ::= <LABEL> : <BLOCK>
<PROGRAM> ::= <BLOCK>
<PROGRAM> ::= <COMPOUND STATEMENT>
<PROGRAM> ::= 'COMMENT'
<LEFT PART> ::= <VARIABLE> :=
<LEFT PART LIST> ::= <LEFT PART>
<LEFT PART LIST> ::= <LEFT PART LIST> <LEFT PART>
<ASSIGNMENT STATEMENT> ::= <LEFT PART LIST> <EXPRESSION>
<GO TO STATEMENT> ::= 'GOTO' <EXPRESSION>
<DUMMY STATEMENT> ::= <EMPTY>
<IF CLAUSE> ::= 'IF' <EXPRESSION> 'THEN'
<IF STATEMENT> ::= <IF CLAUSE> <UNCONDITIONAL STATEMENT>
<CONDITIONAL STATEMENT> ::= <IF STATEMENT>
<CONDITIONAL STATEMENT> ::= <IF STATEMENT> 'ELSE' <STATEMENT>
<CONDITIONAL STATEMENT> ::= <IF CLAUSE> <FOR STATEMENT>
<CONDITIONAL STATEMENT> ::= <LABEL> : <CONDITIONAL STATEMENT>
<FOR LIST ELEMENT> ::= <EXPRESSION>
<FOR LIST ELEMENT> ::= <EXPRESSION> 'STEP' <EXPRESSION>
     'UNTIL' <EXPRESSION>
<FOR LIST ELEMENT> ::= <EXPRESSION> 'WHILE' <EXPRESSION>
<FOR LIST> ::= <FOR LIST ELEMENT>
<FOR LIST> ::= <FOR LIST> , <FOR LIST ELEMENT>
<FOR CLAUSE> ::= 'FOR' <VARIABLE> := <FOR LIST> 'DO'
<FOR STATEMENT> ::= <FOR CLAUSE> <STATEMENT>
<FOR STATEMENT> ::= <LABEL> : <FOR STATEMENT>
```

```
<PARAMETER DELIMITER> ::= ,
<PARAMETER DELIMITER> ::= ) <IDENTIFIER> : (
<ACTUAL PARAMETER> ::= <STRING>
<ACTUAL PARAMETER> ::= <EXPRESSION>
<ACTUAL PARAMETER LIST> ::= <ACTUAL PARAMETER>
<ACTUAL PARAMETER LIST> ::= <ACTUAL PARAMETER LIST>
     <PARAMETER DELIMITER> <ACTUAL PARAMETER>
<IDENTIFIER 1> ::= <IDENTIFIER>
<PROCEDURE DESIGNATOR> ::= <IDENTIFIER 1>
     ( <ACTUAL PARAMETER LIST> )
<PROCEDURE STATEMENT> ::= <PROCEDURE DESIGNATOR>
<PROCEDURE STATEMENT> ::= <IDENTIFIER 1>
<RETURN STATEMENT> ::= 'RETURN' <EXPRESSION>
<SKIP STATEMENT> ::= 'SKIP' <IDENTIFIER> <BLOCK>
<DECLARATION> ::= <TYPE DECLARATION>
<DECLARATION> ::= <ARRAY DECLARATION>
<DECLARATION> ::= <SWITCH DECLARATION>
<DECLARATION> ::= <PROCEDURE DECLARATION>
<DECLARATION> ::= <LABEL DECLARATION>
<DECLARATION> ::= <RESULT DECLARATION>
<DECLARATION> ::= 'DEC' <DENOTATION>
<TYPE LIST> ::= <SIMPLE VARIABLE>
<TYPE LIST> ::= <SIMPLE VARIABLE> , <TYPE LIST>
<TYPE> ::= 'REAL'
<TYPE> ::= 'INTEGER'
<TYPE> ::= 'BOOLEAN'
<LOCAL OR OWN TYPE> ::= <TYPE>
<LOCAL OR OWN TYPE> ::= 'OWN' <TYPE>
<TYPE DECLARATION> ::= <LOCAL OR OWN TYPE> <TYPE LIST>
<LOWER BOUND> ::= <EXPRESSION>
<UPPER BOUND> ::= <EXPRESSION>
<BOUND PAIR HEAD> ::= <LOWER BOUND> :
<BOUND PAIR> ::= <BOUND PAIR HEAD> <UPPER BOUND>
<BOUND PAIR LIST> ::= <BOUND PAIR>
<BOUND PAIR LIST> ::= <BOUND PAIR LIST> , <BOUND PAIR>
<ARRAY SEGMENT> ::= <IDENTIFIER> (/ <BOUND PAIR LIST> /)
<ARRAY SEGMENT> ::= <IDENTIFIER> , <ARRAY SEGMENT>
<ARRAY LIST> ::= <ARRAY SEGMENT>
<ARRAY LIST> ::= <ARRAY LIST> , <ARRAY SEGMENT>
<ARRAY DECLARATION> ::= 'ARRAY' <ARRAY LIST>
<ARRAY DECLARATION> ::= <LOCAL OR OWN TYPE> 'ARRAY' <ARRAY LIST>
<SWITCH LIST> ::= <EXPRESSION>
<SWITCH LIST> ::= <SWITCH LIST> , <EXPRESSION>
<SWITCH DECLARATION> ::= 'SWITCH' <IDENTIFIER> := <SWITCH LIST>
<FORMAL PARAMETER> ::= <IDENTIFIER>
<FORMAL PARAMETER LIST> ::= <FORMAL PARAMETER>
<FORMAL PARAMETER LIST> ::= <FORMAL PARAMETER LIST>
     <PARAMETER DELIMITER> <FORMAL PARAMETER>
<FORMAL PARAMETER PART> ::= ( <FORMAL PARAMETER LIST> )
<IDENTIFIER LIST> ::= <IDENTIFIER>
<IDENTIFIER LIST> ::= <IDENTIFIER LIST> , <IDENTIFIER>
<VALUE PART> ::= 'VALUE' <IDENTIFIER LIST> ;
```

```
<SPECIFIER> ::= 'STRING'
<SPECIFIER> ::= <TYPE>
<SPECIFIER> ::= 'ARRAY'
<SPECIFIER> ::= <TYPE> 'ARRAY'
<SPECIFIER> ::= 'LABEL'
<SPECIFIER> ::= 'SWITCH'
<SPECIFIER> ::= 'PROCEDURE'
<SPECIFIER> ::= <TYPE> 'PROCEDURE'
<SPECIFICATION PART> ::= <SPECIFIER> <IDENTIFIER LIST> ;
<SPECIFICATION PART> ::= <SPECIFICATION PART> <SPECIFIER>
     <IDENTIFIER LIST> ;
<PROCEDURE HEADING> ::= <IDENTIFIER> ;
<PROCEDURE HEADING> ::= <IDENTIFIER> <FORMAL PARAMETER PART> ;
<PROCEDURE HEADING> ::= <IDENTIFIER> <FORMAL PARAMETER PART> ;
<SPECIFICATION PART>
<PROCEDURE HEADING> ::= <IDENTIFIER> <FORMAL PARAMETER PART> ;
     <VALUE PART> <SPECIFICATION PART>
<PROCEDURE BODY> ::= <STATEMENT>
<PROCEDURE DECLARATION> ::= 'PROCEDURE' <PROCEDURE HEADING>
     <PROCEDURE BODY>
<PROCEDURE DECLARATION> ::= <TYPE> 'PROCEDURE'
     <PROCEDURE HEADING> <PROCEDURE BODY>
<LABEL DECLARATION> ::= 'LABEL' <IDENTIFIER>
<RESULT DECLARATION> ::= 'RESULT' <IDENTIFIER>
<RESULT DECLARATION> ::= <TYPE> 'RESULT' <IDENTIFIER>
<SERIAL NUMBER> ::= # <UNSIGNED INTEGER>
<TYPE INDICATOR> ::= <SPECIFIER>
<TYPE INDICATOR> ::= 'FORMAL'
<TYPE INDICATOR> ::= <TYPE> 'RESULT'
<TYPE INDICATOR> ::= 'RESULT'
<DENOTATION> ::= <SERIAL NUMBER> <TYPE INDICATOR>
```

TERMINAL SYMBOL	SYNONYMS
<LOGICAL VALUE>	'TRUE' 'FALSE'
<ADDING OPERATOR>	+ -
<MULTIPLYING OPERATOR>	* / // '/'
'POWER'	**
<RELATIONAL OPERATOR>	= < > <= >= ¬=
	¬> ¬<
	'EQ' 'LS' 'GR' 'LQ' 'GQ'
	'NQ' 'EQUAL' 'LESS'
	'GREATER' 'NOT GREATER'
	'NOT LESS' 'NOT EQUAL'
¬	'NOT'
&	'AND'
|	'OR'
'IMPLIES'	'IMPL'
;	.' $
:	..
:=	..= .=
(	'(1' '(2' '(3' '(4'
	'(5' '(6'
(/	'(/1' '(/2' '(/3'
	'(/4' '(/5' '(/6'
'FOR'	'FOR1' 'FOR2' 'FOR3'
	'FOR4' 'FOR5'
'BEGIN'	'BEGIN1' 'BEGIN2'
	'BEGIN3' 'BEGIN4'
'PROCEDURE'	'PROC'

6.2 Identifier Denotation Examples

In this section are given two examples of the application of the transformation set sequence consisting of the identifier denotation transformation set described in section 3.1.

The first of these examples is a program which demonstrates the application of the various transformations and transformation lists in the transformation set. The action of the <procedure declaration>-transformation list and the appropriate transformations of the <block>-transformation list is illustrated by the denotation of the procedures int proc and properproc and their calls. The former is example 3.1.1 (2), but note that when transformed by the complete transformation set, the procedure body is converted into a skip statement. (Note also the denotation of the value parameters valreal and valarray.) The procedure properproc illustrates the denotation of a label internal to a procedure body which is not a block, while the for statement illustrates the denotation of a label (forinternal) internal to its controlled statement. The remaining declarations illustrate the operation of the other elements of the <block>-transformation list, together with the transformations which convert declarations to single form, and which produce calls to allocate from the information in array and switch declarations. Note that the construction at proglabel1 is example 3.1.1 (1).

The first example program, first in its original form, then after transformation is:

```
'BEGIN'
    'INTEGER' 'PROCEDURE' INTPROC ( VALREAL , VALARRAY , FORMAL ) ;
    'VALUE' VALREAL , VALARRAY ;
    'REAL' VALREAL ;
    'ARRAY' VALARRAY ;
    INTPROC := 'IF' FORMAL <= 0 'THEN'
    1 'ELSE'
    ENTIER ( VALREAL + VALARRAY (/ 1 /) * FORMAL ) + INTPROC ( VALREAL , VALARRAY , FORMAL - 1
) ;
    'PROCEDURE' PROPERPROC ( FORMAL , LABEL ) ;
    'BEGIN'
        PROCINTERNAL : FORMAL := FORMAL - 1 ;
        'GOTO' 'IF' FORMAL <= C 'THEN'
        PROCINTERNAL 'ELSE'
        LABEL
    'END' ;
    'REAL' A ;
    'INTEGER' C , N , M , K ;
    'ARRAY' B , D (/ 1 : 2 /) ;
    'SWITCH' S := PROGLABEL1 , PROGLABEL2 ;
    N := M := K := 10 ;
    'FOR' C := 1 , 2 'DO'
    FORINTERNAL : B (/ C /) := 3 . 14159 / C ;
    C := 3 ;
```

(6.2)

```
    C := INTPROC ( B (/ 2 /) , B , C ) ;
    PROCLABEL1 : 'BEGIN'
        'ARRAY' A (/ 1 : N , 1 : M , - 2 * K : 2 * K /) ;
        A (/ 1 , 1 , 0 /) := 1
    'END' ;
    PROCPERPROC ( C , S (/ 2 /) ) ;
    PROCLABEL2 :
'END'
```

(6.2)

```
'BEGIN'
    'DEC' # 00000929 'INTEGER' 'PROCEDURE' ;
    'DEC' # 00000930 'PROCEDURE' ;
    'DEC' # 00000920 'REAL' ;
    'DEC' # 00000921 'INTEGER' ;
    'DEC' # 00000922 'INTEGER' ;
    'DEC' # 00000923 'INTEGER' ;
    'DEC' # 00000924 'INTEGER' ;
    'DEC' # 00000925 'REAL' 'ARRAY' ;
    'DEC' # 00000926 'REAL' 'ARRAY' ;
    'DEC' # 00000931 'SWITCH' ;
    'DEC' # 00000927 'LABEL' ;
    'DEC' # 00000928 'LABEL' ;
    ALLOCATE ( # 00000931 'SWITCH' , # 00000927 'LABEL' , # 00000928 'LABEL' ) ;
    'SKIP' # 00000930 'PROCEDURE' 'BEGIN'
        'DEC' # 00000914 'RESULT' ;
        'DEC' # 00000915 'LABEL' ;
        'DEC' # 00000916 'FORMAL' ;
        'DEC' # 00000917 'FORMAL' ;
        'BEGIN'
            # 00000915 'LABEL' : # 00000916 'FORMAL' := # 00000916 'FORMAL' - 1 ;
            'GOTO' 'IF' # 00000916 'FORMAL' <= 0 'THEN'
            # 00000915 'LABEL' 'ELSE'
```

```
            # 00000917 'FORMAL'
        'END'
    'END' ;
    'SKIP' # 00000929 'INTEGER' 'PROCEDURE' 'BEGIN'
        'DEC' # 00000908 'INTEGER' 'RESULT' ;
        'DEC' # 00000909 'REAL' 'ARRAY' ;
        'DEC' # 00000910 'REAL' ;
        'DEC' # 00000911 'FORMAL' ;
        'DEC' # 00000912 'FORMAL' ;
        'DEC' # 00000913 'FORMAL' ;
        # 00000910 'REAL' := # 00000911 'FORMAL' ;
        ALLOCATE ( # 00000909 'REAL' 'ARRAY' , # 00000912 'FORMAL' ) ;
        # 00000908 'INTEGER' 'RESULT' := 'IF' # 00000913 'FORMAL' <= 0 'THEN'
        1 'ELSE'
        ENTIER ( # 00000910 'REAL' + # 00000909 'REAL' 'ARRAY' (/ 1 /) * # 00000913 'FORMAL' )
+ # 00000929 'INTEGER' 'PROCEDURE' ( # 00000910 'REAL' , # 00000909 'REAL' 'ARRAY' , # 00000913
 'FORMAL' - 1 ) ;
        'RETURN' # 00000908 'INTEGER' 'RESULT'
    'END' ;
    ALLOCATE ( # 00000926 'REAL' 'ARRAY' , 1 , 2 ) ;
    ALLOCATE ( # 00000925 'REAL' 'ARRAY' , 1 , 2 ) ;
    # 00000922 'INTEGER' := # 00000923 'INTEGER' := # 00000924 'INTEGER' := 10 ;
    'FOR' # C0000921 'INTEGER' := 1 , 2 'DO'
    'BEGIN'
        'DEC' # 00000918 'LABEL' ;
```

(6.2)

```
        # C0000918 'LABEL' : # 00000925 'REAL' 'ARRAY' (/ # 00000921 'INTEGER' /) := 3 . 14159
/ # 00000921 'INTEGER'

    'END' ;

    # 00000921 'INTEGER' := 3 ;

    # 00000921 'INTEGER' := # 00000929 'INTEGER' 'PROCEDURE' ( # 00000925 'REAL' 'ARRAY' (/ 2 /)
 , # 00000925 'REAL' 'ARRAY' , # 00000921 'INTEGER' ) ;

    # 00000927 'LABEL' : 'BEGIN'

        'DEC' # 00000919 'REAL' 'ARRAY' ;

        ALLOCATE ( # 00000919 'REAL' 'ARRAY' , 1 , # 00000922 'INTEGER' , 1 , # 00000923 'INTEGER'
 , - 2 * # 00000924 'INTEGER' , 2 * # 00000924 'INTEGER' ) ;

        # 00000919 'REAL' 'ARRAY' (/ 1 , 1 , 0 /) := 1

    'END' ;

    # C0000930 'PROCEDURE' ( # 00000921 'INTEGER' , # 00000931 'SWITCH' (/ 2 /) ) ;

    # 00000928 'LABEL' :

'END'
```

The second example program is designed to illustrate the implementation of the Algol 60 identifier scope rules. It consists of roughly the same pair of statements, an assignment statement and a go to statement, repeated in various blocks. Some of these blocks are contained in others, and some are parallel to others; all contain redeclarations of some of the identifiers appearing in the pair of statements, thus illustrating the various static scopes for variables and labels. Note that one identifier, Z in the assignment statement between the third and fourth blocks, is used outside the scope of any of its declarations, and hence appears undenoted in the transformed program.

The scope rule demonstration program, first in its original form, and then after transformation, is:

```
'BEGIN'
    'REAL' A ;
    'INTEGER' B ;
    'BOOLEAN' C ;
    'ARRAY' D (/ 1 : 2 /) ;
    'INTEGER' 'ARRAY' E (/ 1 : 2 /) ;
    L1 : A := B + ( 'IF' C 'THEN'
    D (/ 1 /) 'ELSE'
    E (/ 2 /) ) ;
    'IF' 'FALSE' 'THEN'
    'GOTO' L1 ;
    'BEGIN'
        'REAL' A ;
        'INTEGER' B ;
        A := B + ( 'IF' C 'THEN'
        D (/ 1 /) 'ELSE'
        E (/ 2 /) ) ;
        'IF' 'FALSE' 'THEN'
        'GOTO' L1 ;
        'BEGIN'
            'REAL' A ;
            'BOOLEAN' C ;
            'REAL' Z ;
```

```
        L1 : Z := A := B + ( 'IF' C 'THEN'
        D (/ 1 /) 'ELSE'
        E (/ 2 /) ) ;
        'IF' 'FALSE' 'THEN'
        'GOTO' L1
    'END' ;
    L1 : Z := A := B + ( 'IF' C 'THEN'
    D (/ 1 /) 'ELSE'
    E (/ 2 /) ) ;
    'IF' 'FALSE' 'THEN'
    'GOTO' L1 ;
    'BEGIN'
        'REAL' A ;
        'ARRAY' D (/ 1 : 2 /) ;
        'REAL' Z ;
        Z := A := B + ( 'IF' C 'THEN'
        D (/ 1 /) 'ELSE'
        E (/ 2 /) ) ;
        'IF' 'FALSE' 'THEN'
        'GOTO' L1
    'END'
'END' ;
A := B + ( 'IF' C 'THEN'
D (/ 1 /) 'ELSE'
```

```
    E (/ 2 /) ) ;
    'IF' 'FALSE' 'THEN'
    'GOTO' L1
'END'
```

```
'BEGIN'
    'DEC' # 00000902 'REAL' ;
    'DEC' # 00000903 'INTEGER' ;
    'DEC' # 00000904 'BOOLEAN' ;
    'DEC' # 00000905 'REAL' 'ARRAY' ;
    'DEC' # 00000906 'INTEGER' 'ARRAY' ;
    'DEC' # 00000907 'LABEL' ;
    ALLOCATE ( # 00000906 'INTEGER' 'ARRAY' , 1 , 2 ) ;
    ALLOCATE ( # 00000905 'REAL' 'ARRAY' , 1 , 2 ) ;
    # 00000907 'LABEL' : # 00000902 'REAL' := # 00000903 'INTEGER' + ( 'IF' # 00000904 'BOOLEAN'
'THEN'
    # 00000905 'REAL' 'ARRAY' (/ 1 /) 'ELSE'
    # 00000906 'INTEGER' 'ARRAY' (/ 2 /) ) ;
    'IF' 'FALSE' 'THEN'
    'GOTO' # 00000907 'LABEL' ;
    'BEGIN'
        'DEC' # 00000899 'REAL' ;
        'DEC' # 00000900 'INTEGER' ;
        'DEC' # 00000901 'LABEL' ;
        # 00000899 'REAL' := # 00000900 'INTEGER' + ( 'IF' # 00000904 'BOOLEAN' 'THEN'
        # 00000905 'REAL' 'ARRAY' (/ 1 /) 'ELSE'
        # 00000906 'INTEGER' 'ARRAY' (/ 2 /) ) ;
        'IF' 'FALSE' 'THEN'
        'GOTO' # 00000901 'LABEL' ;
```

```
'BEGIN'
    'DEC' # 00000892 'REAL' ;
    'DEC' # 00000893 'BOOLEAN' ;
    'DEC' # 00000894 'REAL' ;
    'DEC' # 00000895 'LABEL' ;
    # 00000895 'LABEL' : # 00000894 'REAL' := # 00000892 'REAL' := # 00000900 'INTEGER'
+ ( 'IF' # 00000893 'BOOLEAN' 'THEN'
    # 00000905 'REAL' 'ARRAY' (/ 1 /) 'ELSE'
    # 00000906 'INTEGER' 'ARRAY' (/ 2 /) ) ;
    'IF' 'FALSE' 'THEN'
    'GOTO' # 00000895 'LABEL'
'END' ;
# 00000901 'LABEL' : Z := # 00000899 'REAL' := # 00000900 'INTEGER' + ( 'IF' # 00000904
'BOOLEAN' 'THEN'
# C0000905 'REAL' 'ARRAY' (/ 1 /) 'ELSE'
# C0000906 'INTEGER' 'ARRAY' (/ 2 /) ) ;
'IF' 'FALSE' 'THEN'
'GOTO' # 00000901 'LABEL' ;
'BEGIN'
    'DEC' # 00000896 'REAL' ;
    'DEC' # 00000898 'REAL' 'ARRAY' ;
    'DEC' # 00000897 'REAL' ;
    ALLOCATE ( # 00000898 'REAL' 'ARRAY' , 1 , 2 ) ;
    # C0000897 'REAL' := # 00000896 'REAL' := # 00000900 'INTEGER' + ( 'IF' # 00000904
'BOOLEAN' 'THEN'
```

(6.2)

```
            # 00000898 'REAL' 'ARRAY' (/ 1 /) 'ELSE'
            # 00000906 'INTEGER' 'ARRAY' (/ 2 /) ) ;
            'IF' 'FALSE' 'THEN'
            'GOTO' # 00000901 'LABEL'
        'END'
    'END' ;
    # 00000902 'REAL' := # 00000903 'INTEGER' + ( 'IF' # 00000904 'BOOLEAN' 'THEN'
    # 00000905 'REAL' 'ARRAY' (/ 1 /) 'ELSE'
    # 00000906 'INTEGER' 'ARRAY' (/ 2 /) ) ;
    'IF' 'FALSE' 'THEN'
    'GOTO' # 00000907 'LABEL'
'END'
```

6.3 For Statement Optimization Examples

In this section are presented three examples of the application of the transformation set sequence consisting of the identifier denotation transformation set followed by the for statement optimization transformation set. The first two examples are intended to illustrate certain aspects of the operation of the for statement optimization transformation set, while the last is intended to show the denotation and optimization of a typical simple Algol program, one which evaluates a polynomial.

The first example program demonstrates the detection of for statements which must not be optimized. Considering the for statements in order, the reasons they may not be optimized are: The first statement (in procedure P) contains a formal parameter. (See the call to P at the end of the program, which gives rise to side effects on i.) In the second, the loop variable is not a simple integer. In the third, the loop variable is assigned in the controlled statement (cf. example 1.1.3(10)). The fourth and fifth for statements contain step-expressions which involve the loop variable and a variable assigned, respectively. The sixth statement (cf. example 1.1.3 (13)) contains a procedure call (which in this case does produce side effects on i). The seventh for statement illustrates the operation of the transformations for multiple for list elements, arithmetic-expression-elements, and while-elements.

The first example program and its transformed form are:

```
'BEGIN'
    'INTEGER' 'PROCEDURE' STEP ;
    STEP := I := I + 1 ;
    'PROCEDURE' P ( X ) ;
    'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
    A (/ I /) := 2 * X ;
    'INTEGER' 'ARRAY' Z (/ 1 : 10 /) ;
    'ARRAY' A (/ 1 : 10 /) ;
    'INTEGER' I , J , K , N ;
    N := 10 ;
    I := J := K := 1 ;
    'FOR' Z (/ I /) := 1 'STEP' 1 'UNTIL' N 'DO'
    A (/ Z (/ I /) /) := 2 * Z (/ I /) ;
    'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
    'BEGIN'
        A (/ I /) := 2 * I ;
        I := I + 1
    'END' ;
    'FOR' I := 1 'STEP' J + I * K 'UNTIL' N 'DO'
    A (/ I /) := I ;
    'FOR' I := 1 'STEP' J + Z (/ 3 /) * K 'UNTIL' N 'DO'
    Z (/ I /) := I ;
    'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
```

(6.3)

```
    'BEGIN'
        A (/ I /) := 2 * I ;
        STEP
    'END' ;
    'FOR' I := 1 , I + 1 'WHILE' I <= N 'DO'
    A (/ I /) := 2 * I ;
    P ( STEP )
'END'
```

(6.3)

```
'BEGIN'
    'DEC' # 00000943 'INTEGER' 'PROCEDURE' ;
    'DEC' # 00000944 'PROCEDURE' ;
    'DEC' # 00000941 'INTEGER' 'ARRAY' ;
    'DEC' # 00000942 'REAL' 'ARRAY' ;
    'DEC' # 00000937 'INTEGER' ;
    'DEC' # 00000938 'INTEGER' ;
    'DEC' # 00000939 'INTEGER' ;
    'DEC' # 00000940 'INTEGER' ;
    'SKIP' # 00000944 'PROCEDURE' 'BEGIN'
        'DEC' # 00000935 'RESULT' ;
        'DEC' # 00000936 'FORMAL' ;
        'BEGIN'
            'DEC' # 00000945 'LABEL' ;
            # 00000937 'INTEGER' := 1 ;
            # 00000945 'LABEL' : 'IF' # 00000937 'INTEGER' <= ( # 00000940 'INTEGER' ) 'THEN'
            'BEGIN'
                # 00000942 'REAL' 'ARRAY' (/ # 00000937 'INTEGER' /) := 2 * # 00000936 'FORMAL'
;
                # 00000937 'INTEGER' := # 00000937 'INTEGER' + 1 ;
                'GOTO' # 00000945 'LABEL'
            'END'
        'END'
    'END' ;
```

(6.3)

```
'SKIP' # 00000943 'INTEGER' 'PROCEDURE' 'BEGIN'
    'DEC' # 00000934 'INTEGER' 'RESULT' ;
    # 00000934 'INTEGER' 'RESULT' := # 00000937 'INTEGER' := # 00000937 'INTEGER' + 1 ;
    'RETURN' # 00000934 'INTEGER' 'RESULT'
'END' ;
ALLOCATE ( # 00000942 'REAL' 'ARRAY' , 1 , 10 ) ;
ALLOCATE ( # 00000941 'INTEGER' 'ARRAY' , 1 , 10 ) ;
# 00000940 'INTEGER' := 10 ;
# 00000937 'INTEGER' := # 00000938 'INTEGER' := # 00000939 'INTEGER' := 1 ;
'BEGIN'
    'DEC' # 00000946 'LABEL' ;
    # 00000941 'INTEGER' 'ARRAY' (/ # 00000937 'INTEGER' /) := 1 ;
    # 00000946 'LABEL' : 'IF' # 00000941 'INTEGER' 'ARRAY' (/ # 00000937 'INTEGER' /) <= (
# 00000940 'INTEGER' ) 'THEN'
    'BEGIN'
        # 00000942 'REAL' 'ARRAY' (/ # 00000941 'INTEGER' 'ARRAY' (/ # 00000937 'INTEGER' /)
/) := 2 * # 00000941 'INTEGER' 'ARRAY' (/ # 00000937 'INTEGER' /) ;
        # 00000941 'INTEGER' 'ARRAY' (/ # 00000937 'INTEGER' /) := # 00000941 'INTEGER' 'ARRAY'
(/ # 00000937 'INTEGER' /) + 1 ;
        'GOTO' # 00000946 'LABEL'
    'END'
'END' ;
'BEGIN'
    'DEC' # 00000947 'LABEL' ;
    # 00000937 'INTEGER' := 1 ;
```

```
        # C0000947 'LABEL' : 'IF' # 00000937 'INTEGER' <= ( # 00000940 'INTEGER' ) 'THEN'
        'BEGIN'
            'BEGIN'
                # 00000942 'REAL' 'ARRAY' (/ # 00000937 'INTEGER' /) := 2 * # 00000937 'INTEGER'
 ;
                # 00000937 'INTEGER' := # 00000937 'INTEGER' + 1
            'END' ;
            # 00000937 'INTEGER' := # 00000937 'INTEGER' + 1 ;
            'GOTO' # 00000947 'LABEL'
        'END'
    'END' ;
    'BEGIN'
        'DEC' # 00000948 'LABEL' ;
        # 00000937 'INTEGER' := 1 ;
        # 00000948 'LABEL' : 'IF' ( # 00000937 'INTEGER' - ( # 00000940 'INTEGER' ) ) * SIGN (
# 00000938 'INTEGER' + # 00000937 'INTEGER' * # 00000939 'INTEGER' ) <= 0 'THEN'
        'BEGIN'
            # C0000942 'REAL' 'ARRAY' (/ # 00000937 'INTEGER' /) := # 00000937 'INTEGER' ;
            # 00000937 'INTEGER' := # 00000937 'INTEGER' + ( # 00000938 'INTEGER' + # 00000937
'INTEGER' * # C0000939 'INTEGER' ) ;
            'GOTO' # 00000948 'LABEL'
        'END'
    'END' ;
    'BEGIN'
        'DEC' # 00000949 'LABEL' ;
        # 00000937 'INTEGER' := 1 ;
```

(6.3)

```
        # CCCC0949 'LABEL' : 'IF' ( # 00000937 'INTEGER' - ( # 00000940 'INTEGER' ) ) * SIGN (
# 00000938 'INTEGER' + # 00000941 'INTEGER' 'ARRAY' (/ 3 /) * # 00000939 'INTEGER' ) <= 0 'THEN'

        'BEGIN'
            # 00000941 'INTEGER' 'ARRAY' (/ # 00000937 'INTEGER' /) := # 00000937 'INTEGER' ;
            # 00000937 'INTEGER' := # 00000937 'INTEGER' + ( # 00000938 'INTEGER' + # 00000941
'INTEGER' 'ARRAY' (/ 3 /) * # 00000939 'INTEGER' ) ;
            'GOTO' # 00000949 'LABEL'
        'END'
    'END' ;
    'BEGIN'
        'DEC' # 00000950 'LABEL' ;
        # 00000937 'INTEGER' := 1 ;
        # CCCC0950 'LABEL' : 'IF' # 00000937 'INTEGER' <= ( # 00000940 'INTEGER' ) 'THEN'
        'BEGIN'
            'BEGIN'
                # 00000942 'REAL' 'ARRAY' (/ # 00000937 'INTEGER' /) := 2 * # 00000937 'INTEGER'
;
                # 00000943 'INTEGER' 'PROCEDURE'
            'END' ;
            # 00000937 'INTEGER' := # 00000937 'INTEGER' + 1 ;
            'GOTO' # 00000950 'LABEL'
        'END'
    'END' ;
    'BEGIN'
        'DEC' # 00000951 'PROCEDURE' ;
```

```
        'SKIP' # 00000951 'PROCEDURE' 'BEGIN'
            'DEC' # 00000952 'RESULT' ;
            # 00000942 'REAL' 'ARRAY' (/ # 00000937 'INTEGER' /) := 2 * # 00000937 'INTEGER'
        'END' ;
        'BEGIN'
            # 00000937 'INTEGER' := 1 ;
            # 00000951 'PROCEDURE'
        'END' ;
        'BEGIN'
            'DEC' # 00000953 'LABEL' ;
            # 00000953 'LABEL' : # 00000937 'INTEGER' := # 00000937 'INTEGER' + 1 ;
            'IF' # 00000937 'INTEGER' <= # 00000940 'INTEGER' 'THEN'
            'BEGIN'
                # 00000951 'PROCEDURE' ;
                'GOTO' # 00000953 'LABEL'
            'END'
        'END'
    'END' ;
    # C000C944 'PROCEDURE' ( # 00000943 'INTEGER' 'PROCEDURE' )
'END'
```

(6.3)

The second example program illustrates various forms of optimizable for statements and the detection of general, linear, and constant subscripted variables within such statements. Considering the for statements in order, the points illustrated are: The first may now be optimized (cf. the previous example) because the parameter X is called by value, hence no side effects can arise. The second illustrates a simple optimizable for statement containing one linear subscripted variable. The third and fourth for statements are optimizable, but in each the until-expression must not be calculated in advance because it contains the loop variable (cf. example 1.1.3 (9)) and a variable assigned (cf. Transformation 9, section 1.1.3), respectively. The fifth statement contains some subscripted variables which are general (not optimizable) because they contain variables assigned (cf. example 1.1.4 (14)). In the sixth statement a subscripted variable is declared within the controlled statement, while in the seventh several subscripted variables have non-linear subscripts; all of these must not be optimized. The eighth statement illustrates that a for statement and subscripted variable containing complicated expressions may still be optimizable.* Finally the last pair of for statements illustrates optimization of nested for statements (cf. the discussion in section 3.2).

The second example program is listed below, first in its original form, and then after transformation. The program is expanded considerably during transformation, largely as a result of for statement optimization.

---

*The redeclaration of serial numbers in the optimized form of this statement (cf. p. 237) arises from the use of *rplacd* to add generable symbols to the environment as discussed in section 2.5.2. The solution proposed

```
'BEGIN'
    'INTEGER' 'PROCEDURE' STEP ;
    STEP := I := I + 1 ;
    'PROCEDURE' P ( X ) ;
    'VALUE' X ;
    'INTEGER' X ;
    'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
    A (/ I /) := 2 * X ;
    'INTEGER' 'ARRAY' Z (/ 1 : 100 , 1 : 100 /) ;
    'ARRAY' A , B , C (/ 1 : 100 /) ;
    'INTEGER' I , J , K , N ;
    'REAL' REAL ;
    REAL := 3 . 14159 ;
    N := 10 ;
    I := J := K := 1 ;
    'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
    A (/ I /) := 2 * I ;
    'FOR' I := 1 'STEP' 1 'UNTIL' N * I - 7 'DO'
    A (/ I /) := 2 * I ;
    'FOR' I := 1 'STEP' + 1 'UNTIL' N * A (/ 1 /) - 7 'DO'
    A (/ I /) := 2 * I ;
    'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
    'BEGIN'
```

(6.3)

```
    J := N - I ;
    C (/ A (/ K /) /) := I ;
    A (/ I /) := B (/ J /)
'END' ;
'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
'BEGIN'
    A (/ I /) := I ;
    'BEGIN'
        'ARRAY' A (/ 1 : I /) ;
        A (/ I /) := B (/ I /)
    'END'
'END' ;
'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
'BEGIN'
    A (/ I * I /) := A (/ I /) ;
    A (/ REAL * I + 1 /) := A (/ 'IF' I < 5 'THEN'
    1 'ELSE'
    I - 3 /)
'END' ;
'FOR' I := 3 * J + ODD ( K ) 'STEP' COS ( REAL ) 'UNTIL' J * N 'DO'
Z (/ ODD ( REAL ) + J , K * I + J /) := I ;
'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
'FOR' J := 1 'STEP' 1 'UNTIL' 2 * I - N 'DO'
Z (/ 1 , 1 /) := Z (/ I , 1 /) + Z (/ 1 , J /) - Z (/ I , J /) ;
```

(6.3)

```
    P ( STEP )
'END'
```

(6.3)

```
'BEGIN'
    'DEC' # CCCC0906 'INTEGER' 'PROCEDURE' ;
    'DEC' # 00000907 'PROCEDURE' ;
    'DEC' # 00000902 'INTEGER' 'ARRAY' ;
    'DEC' # CCCC0903 'REAL' 'ARRAY' ;
    'DEC' # 00000904 'REAL' 'ARRAY' ;
    'DEC' # 00000905 'REAL' 'ARRAY' ;
    'DEC' # 00000897 'INTEGER' ;
    'DEC' # 00000898 'INTEGER' ;
    'DEC' # 00000899 'INTEGER' ;
    'DEC' # 00000900 'INTEGER' ;
    'DEC' # 0000901 'REAL' ;
    'SKIP' # 00000907 'PROCEDURE' 'BEGIN'
        'DEC' # 00000893 'RESULT' ;
        'DEC' # 00000894 'INTEGER' ;
        'DEC' # 00000895 'FORMAL' ;
        # 0000894 'INTEGER' := # 00000895 'FORMAL' ;
        'BEGIN1'
            'DEC' # 00000908 'LABEL' ;
            'DEC' # 00000909 'INTEGER' ;
            'DEC' # 00000910 'INTEGER' ;
            'DEC' # 00000911 'INTEGER' ;
            'BEGIN'
```

(6.3)

```
            # 00000897 'INTEGER' := 1 ;
            # 00000909 'INTEGER' ..= # 00000900 'INTEGER'
        'END' ;
        'BEGIN'
            # 00000910 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) ;

        'END' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
        'BEGIN'
            # 00000911 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) - # 00000910 'INTEGER' ;

        'END' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' - 1 ;
        # 00000908 'LABEL' : 'IF' # 00000897 'INTEGER' <= # 00000909 'INTEGER' 'THEN'
        'BEGIN'
            'VAL' # 00000910 'INTEGER' ..= 2 * # 00000894 'INTEGER' ;
            # 00000910 'INTEGER' ..= # 00000910 'INTEGER' + # 00000911 'INTEGER' ;
            # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
            'GOTO' # 00000908 'LABEL'
        'END'
    'END'
'END' ;
'SKIP' # 00000906 'INTEGER' 'PROCEDURE' 'BEGIN'
```

```
        'DEC' # 00000892 'INTEGER' 'RESULT' ;
        # 00000892 'INTEGER' 'RESULT' := # 00000897 'INTEGER' := # 00000897 'INTEGER' + 1 ;
        'RETURN' # 00000892 'INTEGER' 'RESULT'
    'END' ;
    ALLOCATE ( # 00000905 'REAL' 'ARRAY' , 1 , 100 ) ;
    ALLOCATE ( # 00000904 'REAL' 'ARRAY' , 1 , 100 ) ;
    ALLOCATE ( # 00000903 'REAL' 'ARRAY' , 1 , 100 ) ;
    ALLOCATE ( # 00000902 'INTEGER' 'ARRAY' , 1 , 100 , 1 , 100 ) ;
    # 00000901 'REAL' := 3 . 14159 ;
    # 00000900 'INTEGER' := 10 ;
    # 00000897 'INTEGER' := # 00000898 'INTEGER' := # 00000899 'INTEGER' := 1 ;
    'BEGIN1'
        'DEC' # 00000912 'LABEL' ;
        'DEC' # 00000913 'INTEGER' ;
        'DEC' # 00000914 'INTEGER' ;
        'DEC' # 00000915 'INTEGER' ;
        'BEGIN'
            # 00000897 'INTEGER' := 1 ;
            # 00000913 'INTEGER' ..= # 00000900 'INTEGER'
        'END' ;
        'BEGIN'
            # 00000914 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) ;

        'END' ;
```

(6.3)

```
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
        'BEGIN'
            # 00000915 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) - # 00000914 'INTEGER' ;

        'END' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' - 1 ;
        # 00000912 'LABEL' : 'IF' # 00000897 'INTEGER' <= # 00000913 'INTEGER' 'THEN'
        'BEGIN'
            'VAL' # 00000914 'INTEGER' ..= 2 * # 00000897 'INTEGER' ;
            # 00000914 'INTEGER' ..= # 00000914 'INTEGER' + # 00000915 'INTEGER' ;
            # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
            'GOTO' # 00000912 'LABEL'
        'END'
    'END' ;
    'BEGIN1'
        'DEC' # 00000916 'LABEL' ;
        'DEC' # 00000918 'INTEGER' ;
        'DEC' # 00000919 'INTEGER' ;
        'BEGIN'
            # 00000897 'INTEGER' := 1
        'END' ;
        'BEGIN'
            # 00000918 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) ;
```

```
        'END' ;
        # CCCCC897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
        'BEGIN'
            # 00000919 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) - # 00000918 'INTEGER' ;

        'END' ;
        # C0000897 'INTEGER' ..= # 00000897 'INTEGER' - 1 ;
        # 00000916 'LABEL' : 'IF' # 00000897 'INTEGER' <= ( # 00000900 'INTEGER' * # 00000897 'INTEGER'
- 7 ) 'THEN'
        'BEGIN'
            'VAL' # 00000918 'INTEGER' ..= 2 * # 00000897 'INTEGER' ;
            # 00000918 'INTEGER' ..= # 00000918 'INTEGER' + # 00000919 'INTEGER' ;
            # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
            'GOTO' # 00000916 'LABEL'
        'END'
    'END' ;
    'BEGIN1'
        'DEC' # 00000920 'LABEL' ;
        'DEC' # 00000922 'INTEGER' ;
        'DEC' # 00000923 'INTEGER' ;
        'BEGIN'
            # 00000897 'INTEGER' := 1
        'END' ;
```

```
        'BEGIN'
            # 00000922 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) ;

        'END' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
        'BEGIN'
            # CC000923 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) - # 00000922 'INTEGER' ;

        'END' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' - 1 ;
        # CC000920 'LABEL' : 'IF' # 00000897 'INTEGER' <= ( # 00000900 'INTEGER' * # 00000903 'REAL'
'ARRAY' (/ 1 /) - 7 ) 'THEN'
        'BEGIN'
            'VAL' # 00000922 'INTEGER' ..= 2 * # 00000897 'INTEGER' ;
            # 00000922 'INTEGER' ..= # 00000922 'INTEGER' + # 00000923 'INTEGER' ;
            # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
            'GOTO' # 00000920 'LABEL'
        'END'
    'END' ;
    'BEGIN1'
        'DEC' # 00000924 'LABEL' ;
        'DEC' # 00000925 'INTEGER' ;
        'DEC' # 00000926 'INTEGER' ;
```

```
        'DEC' # 00000927 'INTEGER' ;
        'DEC' # 00000928 'INTEGER' ;
        'BEGIN'
            # 00000897 'INTEGER' := 1 ;
            # 00000925 'INTEGER' ..= # 00000900 'INTEGER'
        'END' ;
        'BEGIN'
            # 00000928 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' (/ # 00000899 'INTEGER' /)
;
            # 00000926 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) ;

        'END' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
        'BEGIN'
            # 00000927 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) - # 00000926 'INTEGER' ;

        'END' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' - 1 ;
        # 00000924 'LABEL' : 'IF' # 00000897 'INTEGER' <= # 00000925 'INTEGER' 'THEN'
        'BEGIN'
            'BEGIN'
                # 00000898 'INTEGER' := # 00000900 'INTEGER' - # 00000897 'INTEGER' ;
                # 00000905 'REAL' 'ARRAY' '(/1' 'VAL' # 00000928 'INTEGER' /) := # 00000897 'INTEGER'
;
```

```
            'VAL' # 00000926 'INTEGER' ..= # 00000904 'REAL' 'ARRAY' '(/1' # 00000898 'INTEGER'
/)
        'END' ;
        # 00000926 'INTEGER' ..= # 00000926 'INTEGER' + # 00000927 'INTEGER' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
        'GOTO' # 00000924 'LABEL'
    'END'
'END' ;
'BEGIN1'
    'DEC' # 00000929 'LABEL' ;
    'DEC' # 00000930 'INTEGER' ;
    'DEC' # 00000931 'INTEGER' ;
    'DEC' # 00000932 'INTEGER' ;
    'DEC' # 00000933 'INTEGER' ;
    'DEC' # 00000934 'INTEGER' ;
    'BEGIN'
        # 00000897 'INTEGER' := 1 ;
        # 00000930 'INTEGER' ..= # 00000900 'INTEGER'
    'END' ;
    'BEGIN'
        # 00000933 'INTEGER' ..= 'REF' # 00000904 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) ;
        # 00000931 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) ;

    'END' ;
```

```
      # CC000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
      'BEGIN'
          # 00000934 'INTEGER' ..= 'REF' # 00000904 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) - # 00CC0933 'INTEGER' ;
          # 00000932 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) - # 00000931 'INTEGER' ;

      'END' ;
      # 00000897 'INTEGER' ..= # 00000897 'INTEGER' - 1 ;
      # 00000929 'LABEL' : 'IF' # 00000897 'INTEGER' <= # 00000930 'INTEGER' 'THEN'
      'BEGIN'
          'BEGIN'
              'VAL' # 00000931 'INTEGER' ..= # 00000897 'INTEGER' ;
              'BEGIN'
                  'DEC' # 0CCCC896 'REAL' 'ARRAY' ;
                  ALLOCATE ( # 00000896 'REAL' 'ARRAY' , 1 , # 00000897 'INTEGER' ) ;
                  # 00000896 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER' /) := 'VAL' # 00000933
'INTEGER'
              'END'
          'END' ;
          # 00000933 'INTEGER' ..= # 00000933 'INTEGER' + # 00000934 'INTEGER' ;
          # CCC00931 'INTEGER' ..= # 00000931 'INTEGER' + # 00000932 'INTEGER' ;
          # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
          'GOTO' # 00000929 'LABEL'
      'END'
```

```
    'END' ;
    'BEGIN1'
        'DEC' # 00000935 'LABEL' ;
        'DEC' # 00000936 'INTEGER' ;
        'DEC' # 00000937 'INTEGER' ;
        'DEC' # 00000938 'INTEGER' ;
        'BEGIN'
            # 00000897 'INTEGER' := 1 ;
            # 00000936 'INTEGER' ..= # 00000900 'INTEGER'
        'END' ;
        'BEGIN'
            # 00000937 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) ;

        'END' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
        'BEGIN'
            # 00000938 'INTEGER' ..= 'REF' # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER'
/) - # 00000937 'INTEGER' ;

        'END' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' - 1 ;
        # 00000935 'LABEL' : 'IF' # 00000897 'INTEGER' <= # 00000936 'INTEGER' 'THEN'
        'BEGIN'
            'BEGIN'
```

(6.3)

```
                # 00000903 'REAL' 'ARRAY' '(/1' # 00000897 'INTEGER' * # 00000897 'INTEGER' /)
:= 'VAL' # 00000937 'INTEGER' ;
                # 00000903 'REAL' 'ARRAY' '(/1' # 00000901 'REAL' * # 00000897 'INTEGER' + 1 /)
 := # 00000903 'REAL' 'ARRAY' '(/1' 'IF' # 00000897 'INTEGER' < 5 'THEN'
                1 'ELSE'
                # 00000897 'INTEGER' - 3 /)
            'END' ;
            # 00000937 'INTEGER' ..= # 00000937 'INTEGER' + # 00000938 'INTEGER' ;
            # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
            'GOTO' # C0000935 'LABEL'
        'END'
    'END' ;
    'BEGIN1'
        'DEC' # 00000939 'LABEL' ;
        'DEC' # 00000940 'INTEGER' ;
        'DEC' # 00000941 'INTEGER' ;
        'DEC' # 00000939 'INTEGER' ;
        'DEC' # 00000940 'INTEGER' ;
        'BEGIN'
            # 00000897 'INTEGER' := 3 * # 00000898 'INTEGER' + COS ( # 00000899 'INTEGER' ) ;
            # 00000940 'INTEGER' ..= COS ( # 00000901 'REAL' ) ;
            # 00000941 'INTEGER' ..= # 00000898 'INTEGER' * # 00000900 'INTEGER'
        'END' ;
        'BEGIN'
            # 00000939 'INTEGER' ..= 'REF' # 00000902 'INTEGER' 'ARRAY' '(/1' COS ( # 00000901
'REAL' ) + # 00000898 'INTEGER' , # 00000899 'INTEGER' * # 00000897 'INTEGER' + # 00000898 'INTEGER'
```

```
/) ;

        'END' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + # 00000940 'INTEGER' ;
        'BEGIN'
            # 00000940 'INTEGER' ..= 'REF' # 00000902 'INTEGER' 'ARRAY' '(/1' COS ( # 00000901
'REAL' ) + # 00000898 'INTEGER' , # 00000899 'INTEGER' * # 00000897 'INTEGER' + # 00000898 'INTEGER'
/) - # 00000939 'INTEGER' ;

        'END' ;
        # 00000897 'INTEGER' ..= # 00000897 'INTEGER' - # 00000940 'INTEGER' ;
        # 00000939 'LABEL' : 'IF' ( # 00000897 'INTEGER' - # 00000941 'INTEGER' ) * SIGN ( # 00000940
'INTEGER' ) <= 0 'THEN'
        'BEGIN'
            'VAL' # 00000939 'INTEGER' ..= # 00000897 'INTEGER' ;
            # 00000939 'INTEGER' ..= # 00000939 'INTEGER' + # 00000940 'INTEGER' ;
            # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + # 00000940 'INTEGER' ;
            'GOTO' # 00000939 'LABEL'
        'END'
    'END' ;
    'BEGIN1'
        'DEC' # 00000950 'LABEL' ;
        'DEC' # 00000951 'INTEGER' ;
        'DEC' # 00000952 'INTEGER' ;
        'DEC' # 00000953 'INTEGER' ;
        'DEC' # 00000954 'INTEGER' ;
```

```
    'BEGIN'
        # 00000897 'INTEGER' := 1 ;
        # 00000951 'INTEGER' ..= # 00000900 'INTEGER'
    'END' ;
    'BEGIN'
        # 00000954 'INTEGER' ..= 'REF' # 00000902 'INTEGER' 'ARRAY' (/ 1 , 1 /) ;
        # 00000952 'INTEGER' ..= 'REF' # 00000902 'INTEGER' 'ARRAY' '(/1' # 00000897 'INTEGER'
, 1 /) ;

    'END' ;
    # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
    'BEGIN'
        # 00000953 'INTEGER' ..= 'REF' # 00000902 'INTEGER' 'ARRAY' '(/1' # 00000897 'INTEGER'
, 1 /) - # 00000952 'INTEGER' ;

    'END' ;
    # 00000897 'INTEGER' ..= # 00000897 'INTEGER' - 1 ;
    # CCCCC950 'LABEL' : 'IF' # 00000897 'INTEGER' <= # 00000951 'INTEGER' 'THEN'
    'BEGIN'
        'BEGIN1'
            'DEC' # 00000942 'LABEL' ;
            'DEC' # 00000943 'INTEGER' ;
            'DEC' # 00000944 'INTEGER' ;
            'DEC' # 00000945 'INTEGER' ;
            'DEC' # 00000946 'INTEGER' ;
```

```
            'DEC' # 00000947 'INTEGER' ;
            'DEC' # 00000948 'INTEGER' ;
            'DEC' # 00000949 'INTEGER' ;
            'BEGIN'
                # 00000898 'INTEGER' := 1 ;
                # 00000943 'INTEGER' ..= 2 * # 00000897 'INTEGER' - # 00000900 'INTEGER'
            'END' ;
            'BEGIN'
                # 00000949 'INTEGER' ..= # 00000952 'INTEGER' ;
                # 00000948 'INTEGER' ..= # 00000954 'INTEGER' ;
                # 00000946 'INTEGER' ..= 'REF' # 00000902 'INTEGER' 'ARRAY' '(/1' # 00000897
'INTEGER' , # 00000898 'INTEGER' /) ;
                # 00000944 'INTEGER' ..= 'REF' # 00000902 'INTEGER' 'ARRAY' '(/1' 1 , # 00000898
'INTEGER' /) ;

            'END' ;
            # 00000898 'INTEGER' ..= # 00000898 'INTEGER' + 1 ;
            'BEGIN'
                # 00000947 'INTEGER' ..= 'REF' # 00000902 'INTEGER' 'ARRAY' '(/1' # 00000897
'INTEGER' , # 00000898 'INTEGER' /) - # 00000946 'INTEGER' ;
                # 00000945 'INTEGER' ..= 'REF' # 00000902 'INTEGER' 'ARRAY' '(/1' 1 , # 00000898
'INTEGER' /) - # 00000944 'INTEGER' ;

            'END' ;
            # 00000898 'INTEGER' ..= # 00000898 'INTEGER' - 1 ;
            # 00000942 'LABEL' : 'IF' # 00000898 'INTEGER' <= # 00000943 'INTEGER' 'THEN'
```

```
                'BEGIN'
                    'VAL' # 00000948 'INTEGER' ..= 'VAL' # 00000949 'INTEGER' + 'VAL' # 00000944
'INTEGER' - 'VAL' # 00000946 'INTEGER' ;
                    # 00000946 'INTEGER' ..= # 00000946 'INTEGER' + # 00000947 'INTEGER' ;
                    # 00000944 'INTEGER' ..= # 00000944 'INTEGER' + # 00000945 'INTEGER' ;
                    # 00000898 'INTEGER' ..= # 00000898 'INTEGER' + 1 ;
                    'GOTO' # 00000942 'LABEL'
                'END'
            'END' ;
            # 00000952 'INTEGER' ..= # 00000952 'INTEGER' + # 00000953 'INTEGER' ;
            # 00000897 'INTEGER' ..= # 00000897 'INTEGER' + 1 ;
            'GOTO' # 00000950 'LABEL'
        'END'
    'END' ;
    # 00000907 'PROCEDURE' ( # 00000906 'INTEGER' 'PROCEDURE' )
'END'
```

(6.3)

The final example is a program for polynomial evaluation, consisting of a procedure which performs the polynomial evaluation and a driver program which reads and prints the data and calls the procedure. Note that the for statement within the procedure is optimizable even though it contains a formal parameter, because this formal parameter is an array identifier, and hence cannot give rise to side effects. The original form of the program, followed by its transformed form, is:

```
'BEGIN'
    'REAL' 'PROCEDURE' EVALUATEPOLYNOMIAL ( F ) OFDEGREE : ( N ) AT : ( X ) ;
    'VALUE' N , X ;
    'ARRAY' F ;
    'REAL' X ;
    'INTEGER' N ;
    'BEGIN'
        'INTEGER' I ;
        'REAL' P ;
        P := F (/ N /) ;
        'FOR' I := N - 1 'STEP' - 1 'UNTIL' 0 'DO'
        P := ( P * X ) + F (/ I /) ;
        EVALUATEPOLYNOMIAL := P
    'END' ;
    'INTEGER' I , N ;
    'REAL' F , X ;
    READ ( N ) ;
    PRINT ( N ) ;
    'BEGIN'
        'ARRAY' A (/ 0 : N /) ;
        'FOR' I := N 'STEP' - 1 'UNTIL' 0 'DO'
        'BEGIN'
            READ ( A (/ I /) ) ;
```

(6.3)

```
            PRINT ( A (/ I /) )
        'END' ;
        READ ( X ) ;
        PRINT ( X ) ;
        F := EVALUATEPOLYNOMIAL ( A , N , X ) ;
        PRINT ( F )
    'END'
'END'
```

(6.3)

```
'BEGIN'
    'DEC' # 00000967 'REAL' 'PROCEDURE' ;
    'DEC' # 00000963 'INTEGER' ;
    'DEC' # C0000964 'INTEGER' ;
    'DEC' # 00000965 'REAL' ;
    'DEC' # 00000966 'REAL' ;
    'SKIP' # C0000967 'REAL' 'PROCEDURE' 'BEGIN'
        'DEC' # 00000956 'REAL' 'RESULT' ;
        'DEC' # 00000957 'INTEGER' ;
        'DEC' # 00000958 'REAL' ;
        'DEC' # 00000959 'FORMAL' ;
        'DEC' # 00000960 'FORMAL' ;
        'DEC' # 00000961 'FORMAL' ;
        # C0000958 'REAL' := # C0C00961 'FORMAL' ;
        # 00000957 'INTEGER' := # 00000960 'FORMAL' ;
        'BEGIN'
            'DEC' # 00000954 'INTEGER' ;
            'DEC' # 00000955 'REAL' ;
            # C0000955 'REAL' := # 00000959 'FORMAL' (/ # 00000957 'INTEGER' /) ;
            'BEGIN1'
                'DEC' # 00000968 'LABEL' ;
                'DEC' # 00000969 'INTEGER' ;
                'DEC' # 00000970 'INTEGER' ;
```

```
            'DEC' # 00000971 'INTEGER' ;
            'BEGIN'
                # 00000954 'INTEGER' := # 00000957 'INTEGER' - 1 ;
                # 00000969 'INTEGER' ..= 0
            'END' ;
            'BEGIN'
                # 00000970 'INTEGER' ..= 'REF' # 00000959 'FORMAL' '(/1' # 00000954 'INTEGER'
/) ;

            'END' ;
            # 00000954 'INTEGER' ..= # 00000954 'INTEGER' - 1 ;
            'BEGIN'
                # 00000971 'INTEGER' ..= 'REF' # 00000959 'FORMAL' '(/1' # 00000954 'INTEGER'
/) - # 00000970 'INTEGER' ;

            'END' ;
            # 00000954 'INTEGER' ..= # 00000954 'INTEGER' + 1 ;
            # 00000968 'LABEL' : 'IF' # 00000954 'INTEGER' >= # 00000969 'INTEGER' 'THEN'
            'BEGIN'
                # 00000955 'REAL' := ( # 00000955 'REAL' * # 00000958 'REAL' ) + 'VAL' # 00000970
'INTEGER' ;
                # 00000970 'INTEGER' ..= # 00000970 'INTEGER' + # 00000971 'INTEGER' ;
                # 00000954 'INTEGER' ..= # 00000954 'INTEGER' - 1 ;
                'GOTO' # 00000968 'LABEL'
            'END'
        'END' ;
```

(6.3)

```
        # 00000956 'REAL' 'RESULT' := # 00000955 'REAL'
    'END' ;
    'RETURN' # 00000956 'REAL' 'RESULT'
'END' ;
READ ( # 00000964 'INTEGER' ) ;
PRINT ( # 00000964 'INTEGER' ) ;
'BEGIN'
    'DEC' # 00000962 'REAL' 'ARRAY' ;
    ALLOCATE ( # 00000962 'REAL' 'ARRAY' , 0 , # 00000964 'INTEGER' ) ;
    'BEGIN1'
        'DEC' # 00000972 'LABEL' ;
        'DEC' # 00000973 'INTEGER' ;
        'DEC' # 00000974 'INTEGER' ;
        'DEC' # 00000975 'INTEGER' ;
        'BEGIN'
            # 00000963 'INTEGER' := # 00000964 'INTEGER' ;
            # 00000973 'INTEGER' ..= 0
        'END' ;
        'BEGIN'
            # 00000974 'INTEGER' ..= 'REF' # 00000962 'REAL' 'ARRAY' '(/1' # 00000963 'INTEGER'
/) ;

        'END' ;
        # 00000963 'INTEGER' ..= # 00000963 'INTEGER' - 1 ;
```

(6.3)

```
            'BEGIN'
                # 00000975 'INTEGER' ..= 'REF' # 00000962 'REAL' 'ARRAY' '(/1' # 00000963 'INTEGER'
 /) - # 00000974 'INTEGER' ;

            'END' ;
            # 00000963 'INTEGER' ..= # 00000963 'INTEGER' + 1 ;
            # 00000972 'LABEL' : 'IF' # 00000963 'INTEGER' >= # 00000973 'INTEGER' 'THEN'
            'BEGIN'
                'BEGIN'
                    READ ( 'VAL' # 00000974 'INTEGER' ) ;
                    PRINT ( 'VAL' # 00000974 'INTEGER' )
                'END' ;
                # 00000974 'INTEGER' ..= # 00000974 'INTEGER' + # 00000975 'INTEGER' ;
                # 00000963 'INTEGER' ..= # 00000963 'INTEGER' - 1 ;
                'GOTO' # 00000972 'LABEL'
            'END'
        'END' ;
        READ ( # 00000966 'REAL' ) ;
        PRINT ( # 00000966 'REAL' ) ;
        # 00000965 'REAL' := # 00000967 'REAL' 'PROCEDURE' ( # 00000962 'REAL' 'ARRAY' , # 00000964
 'INTEGER' , # 00000966 'REAL' ) ;
        PRINT ( # 00000965 'REAL' )
    'END'
'END'
```

B